panikzettel.htwr-aachen.de

Funktionale Programmierung

Version 0 |

 Jonas Maximus Schneider Leo std::vector Werner Destina Kolac <3

 Adrian Jakob Groh Peter (Peti) Wallmeyer :) Tore Ulrich Kunze

Inhaltsverzeichnis

1 Einleitung und Vorwissen 1

2 Haskell 2

- 2.2 Operatorrangfolge 3
- 2.3 Haskell 101 4
- 2.4 Funktionen höherer Ordnung 5
- 2.5 Programmieren mit Lazy Evaluation 6
- 2.6 Monaden 6

3 Denotationelle Semantik funktionaler Programme 7

- 3.1 Ordnungen 8
- 3.2 Fixpunkte 10
- 3.3 Die Haskell Semantik 10
- 3.4 Reduzierung von *Haskell* zu *Simple Haskell* 11

4 Der Lambda-Kalkül 12

- 4.1 Lambda-Terme 12
- 4.2 Reduktionsregeln 13
- 4.3 Übersetzung von einfachem Haskell in Lambda-Terme 13

5 Typüberprüfung und -inferenz 14

1 Einleitung und Vorwissen

Vorwort

Dieser Panikzettel ist Open Source auf `https://git.rwth-aachen.de/jonas.max.schneider/panikzettel` . Wir freuen uns über Anmerkungen und Verbessungsvorschläge (auch von offiziellen Quellen).

Vorwissen

Praktisch keines. Programmierung könnte nicht schaden, aber es wird hier alles von 0 aufgebaut.

2 Haskell

Haskell Programmierung ist ein wesentlicher Teil der Vorlesung. Wir trennen zwischen der Auswertungstrategien & Sprachkonstrukte und wesentlichen Funktionen, damit sich ein übersichtlicher Panikzettel ergibt.

Typdeklarationen

```
-- Dies ist ein Kommentar
square :: Int -> Int -- Also ein Funktion von int zu int
```

Funktionsdeklarationen

```
square x = x * x
```

Auswertungsstrategien

Zu unterscheiden sind *strikte* Auswertungsstrategien, bei denen immer der innerste Ausdruck als nächstes ausgewertet wird (*innermost evaluation*), und *nicht-strikte*, bei denen der äußerste Ausdruck zuerst evaluiert wird.

Haskell verwendet eine Abwandlung einer *nicht-strikten* Auswertung, da es Ausdrücke nur einmal auswertet. Diese *leftmost outermost* evaluation heißt auch *Lazy Evaluation* (da es immer nur das ausführt was es gerade braucht).

Das Ergebnis ist unabhängig von der Auswertungsstrategie gleich. Wenn irgendeine Auswertungsstrategie terminiert, terminiert auch die nicht-strikte Auswertung, aber nicht unbedingt die strikte Auswertung.

Currying

Statt

```
plus :: (Int, Int) -> Int
plus (x,y) = x + y
```

schreibe

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Somit erschafft man im Zwischenschritt der Auswertung von `plus 1 2` die Funktion `plus1 y = 1 + y`, da man zuerst eine Funktion von `Int -> Int` zurück gibt.

Pattern Matching

Die Argumente auf der linken Seite einer Gleichung können statt Variablen auch *Patterns* sein.

```
und :: Bool -> Bool -> Bool
und True y = y
und False y = False

len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

Die Patterns werden von oben nach unten durchgegangen und die erste passende Gleichung wird angewendet.

Lokale Deklarationen

Mit `where` können lokale Deklarationsblöcke erstellt werden.

```haskell
roots a b c = ((-b - d)/e, (-b + d)/e)
      where d = sqrt (b*b - 4*a*c)
            e = 2*a
```

2.2 Operatorrangfolge

Welche von den folgenden Operatoren priorisiert wird, kann durch die Bindungspriorität (1-9) festgelegt werden bsp. `infixl 9 %%`.

Präfix

Ist das normale z.B. `plus 1 2`, Infixfunktionen können mittels Klammerung als Präfix-Funktionen verwendet werden.

```haskell
(+) 1 2 = 1 + 2
```

Infix

Analog können Präfix-Funktionen mit Backquotes zu Infix-Operatoren umgewandelt werden.

```haskell
15 `mod` 4 = mod 15 4
```

Bei Infix-Operatoren muss allerdings festgelegt werden, zu welcher Seite sie *assoziieren*.

```haskell
infixl `divide`
(36 `divide` 6) `divide` 2  -- 3

infixr `divide`
(36 `divide` 6) `divide` 2  -- 12
```

- Der Funktionsraumkonstruktor `->` assoziiert nach rechts.
- Die Funktionasanwendung assoziiert nach links.

Sidenote: Postfix

gibt es nicht.

Ausdrücke

Dies ist nur eine kleine Auswahl der verfügbaren Ausdrücke in Haskell, für eine ausführlichere Liste, siehe 1.1.2 im Skript.

```haskell
-- Listen
[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

-- if then else
if exp1 then exp2 else exp3

-- Guards
maxi (x,y) | x >= y =x
           | otherwise = y

-- let in
roots a b c = let d = sqrt (b*b - 4*a*c)
                  e = 2*a
              in ((-b - d)/e, (-b + d)/e)

-- case of
und x y = case x
            of True -> y
               False -> False
```

```
-- lambda Funktionen
plus = \x y -> x + y
plus x = \y -> x + y
```

Patterns

Für die Argumente in Funktionsdeklarationen werden sog. *Patterns* angegeben, mit der die Form der erlaubten Argumente bestimmt wird. Argumente müssen nur solange ausgewertet werden, bis eines der angegebenen Patterns passend ist.

```
zeros :: [Int]
zeros = 0 : zeros

f :: [Int] -> [Int] -> [Int]
f [] _ = []
f _ [] = []

f [] zeros -- terminiert, obwohl zeros nicht terminiert
```

Patterns müssen *linear* sein, d.h. dass keine Variable in einem Pattern mehrfach vorkommen darf.

Das Zeichen _ ist der Joker-Pattern, der auf jeden Wert passt, aber keine Variable darauf bindet. Daher darf er auch mehrfach in einem Pattern vorkommen.

Mit `@` kann der zu matchende Ausdruck zusätzlich an eine Variable gebunden werden.

```
f y@(x : xs) = x : y  -- kopiert das erste Element
```

2.3 Haskell 101

Typen

```
Int,
Integer, -- (Big Int)
Bool,
Float,
Double,
Char,
a -> b -- Funktionen aller Art
(Int, Int) -- Tupel
[Int] -- Liste von Ints
```

Parametrische Polymorphie

Funktionen, die auf Variablen von verschiedenen Typen angewendet werden können

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Typdefinitionen

```
type Position = (Float, Float)
type String = [Char]

data Color = Red | Yellow | Green
data Nats = Zero | Succ Nats
data List a = Nil | Cons a (List a)
```

Typklassen

Typen können einen *Kontext* haben, z.B. `Eq a`.

```
(==), (/=) :: Eq a => a -> a -> Bool
```

bedeutet, dass für die Typvariable `a` nur Typen aus der Typklasse `Eq` eingesetzt werden dürfen.

```
class Eq where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y) -- wir definieren hier also /=, somit muss a nur noch == selber definieren

instance Eq a => Eq [a] where -- also wenn a zu Eq gehört, tut es auch [a]
  [] == [] = True
  (x:xs) == (y:ys) = x ==y && xs == ys
  _ == _ = False
```

Unter anderem die Klassen `Eq` und `Ord` sind vordefiniert, wobei `Ord` eine *Unterklasse* von `Eq` ist.

```
class Eq a => Ord a where  -- nur wenn a zu Eq gehört, kann a auch zu Ord gehören
(<),(>),(<=),(>=) :: a -> a -> Bool
x < y = x <= y && x /= y
-- ...
```

Um eine Variable einer Datenstruktur auszugeben, brauchen diese eine Implementierung der Methode `show`, welche durch `deriving Show` automatisch erzeugt werden kann. Sonst kann diese wie folgt selbst deklariert werden:

```
instance Show a => Show (List a) where -- erneut, wenn a in Show ist, definieren wir hier Show für [a]
  show Nil = "[]"
  show (Cons x xs) = show x ++ " : " ++ show xs
```

2.4 Funktionen höherer Ordnung

Dies sind Funktionen, dessen Argumente oder Resultat ebenfalls Funktionen sind.

Einfach mal runtergerattert.

```
-- Funktionskomposition .
half . square -- erst x*x dann y/2 => (x*x)/2

-- curry und uncurry
-- Mit den Funktionen `curry` und `uncurry` ist es möglich, eine Funktion beliebig als curried oder uncurried Variante zu verwenden.
uncurry (+) (1,2) -- 3

-- Map
map (*2) [1..10] = [2, 4, 6, 8, .., 20] -- wendet eine Funktion auf alle Element einer Liste an

-- filter
filter even [1..10] = [2,4, .., 10] -- Behält nur die Elemente einer Liste, für die die Funktion true zurückgibt.

-- zipWith
-- Mit zipWith lassen sich zwei Listen mit Anwendung einer Funktion vereinen.
zipWith (+) [1,2] [3,4]  -- [4,6]

-- Fold
-- Mit foldl und foldr können alle Elemente einer Liste auf ein Element "gefaltet" werden, indem sie mit einer Funktion mit dem Accumulator verrechnet werden.
```

```haskell
foldr (+) 0 [1..10]  -- 55 (wie sum)

concat :: [[a]] -> [a]
concat = foldr (++) []
```

Listenkomprehension

```haskell
[ x*x | x <- [1..5], odd x]
```

Listenkokmprehensionen können mit der *Generatorregel* und der *Einschränkungsregel* ausgewertet werden.

Ein Generator hat die Form `var <- exp` (z.B. `x <- [1..5]`). Eine Einschränkung ist dann ein Wahrheitswert z.B. `even x`.

Wir bauen also zuerst alle möglichen Werte mit Generatoren und schränken sie danach ein, um unsere Liste zu fertigen.

2.5 Programmieren mit Lazy Evaluation

```haskell
infinity :: Int
infinity = infinity + 1

mult :: Int -> Int -> Int
mult 0 y = 0
mult x y = x * y
```

Die Auswertung von `mult 0 infinity` terminiert mit dem Ergebnis 0, während `mult infinity 0` nicht terminiert.

Dank lazy evaluation können auch unendliche Datenstrukturen wie die Liste `[5..]` praktisch sein, da diese nur so weit wie nötig ausgewertet werden. Mit der Funktion `take` können z.B. die ersten n Elemente zurückgegeben werden.

```haskell
take 2 [5..]  -- [5,6]
```

Mit der Funktion `takeWhile` können solange Werte aus einer Liste genommen werden, wie eine Bedingung erfüllt ist.

```haskell
takeWhile (<100) [x^2 | x <- [5..]]  -- [25,36,49,64,81]
```

Zyklische Datenobjekte

Datenobjekte, wie `[5,5,5,5,...]`, können effizient abgespeichert werden, indem Zykel genutzt werden. Somit können manche Probleme effizienter gelöst werden (z.B. Hamming, siehe Skript.)

```haskell
fives :: [Int]
fives = 5 : fives
```

2.6 Monaden

Ein- und Ausgabe mit Monaden

Rein funktionale Programmiersprachen haben keine Seiteneffekte. Diese können aber bei z.B. der Interaktion mit der Umwelt gewünscht sein.

Für Ausgabe kann daher der vordefinierte Datentyp `IO ()` benutzt werden, dessen Werte *Aktionen* sind. Die Auswertung eines Ausdrucks vom Typ `IO ()` bedeutet dann, dass die Aktion ausgeführt wird.

```haskell
putChar :: Char -> IO ()
putChar '!'  -- gibt '!' aus

(>>) :: IO () -> IO () -> IO ()  -- "then", Kombination von Aktionen
```

```
putChar 'a' >> putChar 'b'  -- gibt "ab" aus
```

Zur Eingabe wird der Typ `IO a` verwendet, von Aktionen, die jeweils einen Wert von Typ `a` berechnen.

```
getChar :: IO Char

return :: a -> IO a

return '!'  -- Aktion, die nichts tut und bei der sich das Zeichen '!' ergibt
```

Mit `>>=` („bind“) kann der von einer Aktion bestimmte Wert von der nächsten Aktion benutzt werden.

```
echo :: IO ()
echo getChar >>= putChar
```

Da die Notation teilweise schlecht lesbar ist, wird eine neue Notation mit `do` eingeführt.

```
p >>= \x ->
      q >>= \y ->
            r
-- ==
do x <- p
   y <- q
   r
```

Programmieren mit Monaden

Monaden können wie folgt definiert werden:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  ...
```

`Monad` ist keine Typklasse, sondern eine *Konstruktorklasse* (weil `m` ein einstelliger Typkonstruktor ist).

Monaden dienen zur Kapselung von Werten, d.h. in Objekten vom Typ `m a` ist ein Objekt vom Typ `a` gekapselt. Der Typkonstruktor `IO` ist ein Beispiel für eine Monade, d.h. die folgende Instanzdeklaration ist in der Prelude enthalten:

```
instance Monad IO where ...
```

Die Funktionen `return` und `>>=` sind Methoden der Klasse `Monad`, müssen also für jede Instanz dieser Klasse implementiert werden. Die `do`-Notation dagegen ist für alle Instanzen der Klasse `Monad` definiert.

Die Folgenden Gesetze müssen für `return` und `>>=` erfüllt sein, damit ein Typkonstruktor `m` als Monade bezeichnet wird:

1. `p >>= return  =  p`
2. `return x >>= f  =  f x`
3. `p >>= \x -> q >>= \y -> r  =  p >>= \x -> (q >>= \y -> r)`, falls `x` nicht in `r` auftritt

3 Denotationelle Semantik funktionaler Programme

Ab hier wird es wesentlich theoretischer. Wir wollen nun versuchen die Semantik von Haskell zu definieren. Hierfür gibt es 3 Arten diese zu definieren:

Denotationelle Semantik Hier weisen wir jedem Ausdruck ein Mathematisches Äquivalent zu, was dieses definiert

Operationelle Semantik Hier abstrahieren wir meist ein Programm und geben einen abstrakten Interpreter an, welcher das Ergebnis festlegt.

Axiomatische Semantik Werden wir nicht behandeln.

PS. die meisten Sprachen geben einfach nur eine Referenz Implementierung eines Compilers.

Wir fangen mit der denotationellen an und werden später auch noch operationelle behandeln.

—

Ziel wird es also sein, eine Funktion $\mathcal{V}al(\text{exp})$, die jedem Haskell Ausdruck ein mathematisches Objekt zuordnet. . z.B. wird die Haskell `5` auf die mathematische 5 abgebildet und

```
square :: Int -> Int
square x = x * x
```

auf eine mathematische Funktion, die Zahlen quadriert. Da es in Haskell aber auch Ausdrücke gibt, die nicht oder nur partiell definiert sind, wie

```
non_term :: Int -> Int
non_term x = non_term (x + 1)
```

gibt es auch den undefinierten Wert $\perp$ (bottom).

3.1 Ordnungen

Für die Übertragung brauchen wir Funktionen mit bestimmten Eigenschaften. Hierfür definieren wir uns eine partielle Ordnung, die besagt „wie stark“ eine Sache definiert ist.

Eine Ordnung ist, genau das was man vermutet:
- $x \leq x$, Reflexivität
- $x \leq y \land y \leq x \Rightarrow x = y$, Antisymmetrie
- $x \leq y \land y \leq z \Rightarrow x < z$, Transitivität

Bei einer partiellen Ordnung müssen aber nicht alle Element miteinander in Relation stehen. Es könnte also sein, dass $x_1 \leq x_2 \land x_1 \leq x_3$ aber $x_2 \not\leq x_3 \land x_3 \not\leq x_2$. x_2 und x_3 sind also einfach nicht vergleichbar.

Eine Verbund $x_1 \leq x_2 \leq x_3 \leq \ldots, \quad x_1, x_2, x_3, \ldots \in D$ heißt *Kette*.

Wir bauen unsere partielle Ordnung auf der Definiertheit der Werte auf und geben ihr das Zeichen $\sqsubseteq$.

Domains
Für jeden Haskell Typ bauen wir eine eigene Menge von Mathematischen Objekten, genannt *Domains*, sodass jedem Audruck des Typs ein Objekt aus der Domain zugeordnet wird.

Für den Typ Integer wählen wir den *Domain* $\mathbb{Z}_\perp = \left\{\perp_{\mathbb{Z}_\perp}, 0, 1, -1, 2, -2, \ldots\right\}$ für Booleans $\mathbb{B}_\perp = \left\{\perp_{\mathbb{B}_\perp}, \text{True}, \text{False}\right\}$, etc.

Die Struktur eines Domains D wird durch eine partielle Ordnung $\sqsubseteq_D$ festgelegt, die ihre Elemente danach ordnet, wie „stark sie definiert sind“. $x \sqsubseteq_D y$ bedeutet also, dass x höchstens so sehr definiert ist wie y.

> Wieso denn nicht nur ein $\perp$?

Da in Haskell auch korrekt getypte Ausdrücke nicht-terminieren können (so wie in allen Turing-Complete Sprachen), muss die Semantik also auch den Befriff eines undefinierten Wertes für einen Typen wiederspiegeln. Hierfür nehmen wir $\perp_D$ für einen Typ D. Es gilt für alle $y \in D: \quad \perp_D \sqsubseteq_D y$. Für $\mathbb{Z}_\perp$ oder $\mathbb{B}_\perp$ scheint das noch nicht besonders praktisch zu sein, aber...

Werte können nicht nur definiert und undefiniert sein, z.B. können Tupel als einzelne Komponenten den undefinierten Wert haben und damit weniger definiert sein, als voll definierte.

D heißt ein Basis-Domain, wenn nur $\perp_D$ mit den anderen Element in Relation stehen. Alle anderen sind „gleichwertig“ (**nicht gleich** $1 \neq 2$, aber sie stehen nicht in der Ordnung $1 \not\sqsubseteq 2 \vee 2 \not\sqsubseteq 1$). Z.B. $\mathbb{B}_\perp, \mathbb{Z}_\perp, ...$

Ja leider *ein* Domain, genauer *ein Scott-Ershov Domain* .

Tupel von Domains sind erneut Domains. $(d_1, ..., d_n) \sqsubseteq_{D_1 \times ... \times D_n} (d'_1, ..., d'_n)$ gilt genau dann wenn (gdw.) $d_i \sqsubseteq d'_i$ für alle $1 \leq i \leq n$ (Zu beachten ist der Typ/Subscript der Relation). Somit ist $\left(\perp_{D_1}, ..., \perp_{D_n}\right) = \perp_{D_1 \times ... \times D_n}$ das kleinste Element.

Für Funktionen $f, g : D_1 \to D_2$ gilt $f \sqsubseteq_{D_1 \to D_2} g$ gdw. für alle $d \in D_1 : f(d) \sqsubseteq_{D_2} g(d)$. $\perp_{D_1 \to D_2} = d \Rightarrow \perp_{D_2}$ ist somit das kleinste Element.

Definition: Extension von Funktionen

Sei $f : A \to B$ eine Funktion. Jede Funktion $f' : A_\perp \to B$ bezeichnet man als Extension von f wobei $A_\perp = A \cup \left\{\perp_{A_\perp}\right\}$ und wobei $f(d) = f'(d)$ für alle $d \in A$.

Definition: Striktheit von Funktionen

Eine Funktion $g : D_1 \times ... \times D_n \to D$ für Domains $D_1, ..., D_n, D$ heißt *strikt*, falls $g(d_1, ..., d_n) = \perp_D$ für alle $d_1, ..., d_n$ gilt, bei denen ein $d_i = \perp_{D_i}$ ist. Ansonsten heißt g *nicht-strikt*.

Definition: Monotone Funktionen

Seien $\sqsubseteq_{D_1}$ und $\sqsubseteq_{D_2}$ partielle Ordnungen auf D_1 bzw. D_2. Eine Funktion $f : D_1 \to D_2$ ist monoton $\Leftrightarrow \forall d \sqsubseteq_{D_1} d' : f(d) \sqsubseteq_{D_2} f(d')$

Satz: Aus Striktheit folgt Monotonie

Seien $D_1, ..., D_n, D$ Domains, wobei $D_1, ..., D_n$ flach sind. Sei $f : D_1 \times ... \times D_n \to D$ strikt. Dann ist f auch monoton.

Wir definieren für jede Menge D mit p. Ordnung $\sqsubseteq_D$ den Lift von D, $D_\perp$ so, dass $\perp_{D_\perp} \sqsubseteq_{D_\perp} d$ kleiner ist als alle $d \in D$ (inklusive, des eventuellen Vorherigen $\perp_D$). Alle anderen Elemente der Relationen werden beibehalten.

Eigenschaften von $\sqsubseteq$

Dass $\sqsubseteq$ eine partielle Ordnung ist, ist offensichtlich.

Definition: Kleinste obere Schranke (Supremum)

Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge D und S eine Teilmenge von D. Ein Element $d \in D$ ist eine *obere Schranke* von S, falls $d' \sqsubseteq d$ für alle $d' \in S$ gilt. Das Element d ist die *kleinste obere Schranke* (oder „Supremum“), falls darüber hinaus für alle anderen oberen Schranken e der Zusammenhang $d \sqsubseteq e$ gilt. Wir bezeichnen dieses Element als $\sqcup S$

Aber nicht nur das, für jede Domain D:
- Gibt es ein kleinstes Element $\bot_D$.
- Jede Kette $S = d_1 \sqsubseteq d_2 \sqsubseteq ... \sqsubseteq d_n \in D$ hat ein Supremum $\sqcup S$.

Dank diesen zwei Eigenschaften heißt $\sqsubseteq_D$ eine vollständige Partielle Ordnung. Dies gilt für Basis-Domains, Tuple-Domains und Funktionsdomains (Satz 2.1.13)

3.2 Fixpunkte

Definition: Fixpunkt

f ist ein Fixpunkt von ff gdw.. $ff(f) = f$

Der kleinste Fixpunkt (welcher am wenigsten definiert ist) wird als least fixpoint (lfp) bezeichnset.

Satz: Fixpunktsatz

Sei $\sqsubseteq$ eine vollständige partielle Ordnung auf D und sei $f : D \to D$ stetig. Dann besitzt f einen kleinsten Fixpunkt und es gilt $\text{lfp}\, f = \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$.

Funktionen wie ff, die aus Haskell-Ausdrücken gewonnen werden, sind immer stetig. Insofern ist durch den Fixpunktsatz garantiert, dass ihr kleinster Fixpunkt existiert und durch die kleinste obere Schranke der Kette $\left\{\bot, ff(\bot), ff^2(\bot), ...\right\}$ erhalten werden kann. Insgesamt fassen wir also in der denotationellen Semantik die Definition einer Funktion als eine *(Fixpunkt)gleichung* über Funktionen auf. Da hier Gleichungen immer lösbar sind, erhalten wir mindestens eine Funktion als Lösung. Unter allen Lösungen wird dann die kleinste ausgewählt und dem Funktionssymbol als Semantik zugeordnet.

Definition: Stetigkeit

Sei $\sqsubseteq_{D_1}$ vollständig auf D_1 und $\sqsubseteq_{D_2}$ vollständig auf D_2. Eine Funktion $f : D_1 \to D_2$ heißt *stetig* (*continuous*) gdw.. für jede Kette S von D_1 jeweils $f(\sqcup S) = \sqcup\{f(d) \mid d \in S\}$ ist. Die Menge aller stetigen Funktionen von D_1 nach D_2 wird mit $\langle D_1 \to D_2 \rangle$ bezeichnet.

Satz: Stetigkeit und Monotonie

Seien $\sqsubseteq_{D_1}$ und $\sqsubseteq_{D_2}$ vollständige Ordnungen auf D_1 bzw. D_2 und sei $f : D_1 \to D_2$ eine Funktion.
1. f ist stetig gdw.. f monoton ist und für jede kette S von D_1 gilt $f(\sqcup S) \sqsubseteq \sqcup f(S)$.
2. Falls D_1 nur endliche Ketten besitzt, dann ist f stetig gdw. f monoton ist.

Sidenote: Relevanz?

Ja leider haben die Übungsblätter einige Aufgaben hierzu enthalten.

3.3 Die Haskell Semantik

Einfaches Haskell

Um die Semantik von Haskell zu definieren ist es hilfreich, sich auf das Subset *simple Haskell* zu konzentrieren und den Rest dann darauf zu reduzieren.

Ein *simple* Haskell Programm verwendet keine vordefinierten Listen, Typabkürzungen, Typklassen und Pattern matching und ist nur in einer einzigen Deklaration geschrieben.

Es sind somit noch folgende Konstrukte erlaubt:

- var
- Konstante
- Int / Integer
- Float / Double
- Char
- $(\underline{\texttt{exp}}_1, ..., \underline{\texttt{exp}}_n), \quad n \geq 1$
- if $\underline{\texttt{exp}}_1$ then $\underline{\texttt{exp}}_2$ else $\underline{\texttt{exp}}_3$
- let var = $\underline{\texttt{exp}}$ in $\underline{\texttt{exp}}'$
- var -> exp
- Ein paar vordefinierte Funktionen in der Umgebung ω (z.B. <=, -, $*$)

Sidenote: Umgebungen / Environments

Eine Umgebung $\rho : \mathsf{Var} \to \mathsf{Dom}$ ist eine Variablenbelegung, genauer eine partielle Funktion, die Werte auf ein Domain Dom. ρ ist nur für endlich viele Variablen definiert.

Wir brauchen nun noch den Begriff der *freien* Variablen $\text{free}\big(\underline{\texttt{exp}}\big)$, er ordnet jedem *simple Haskell* Ausdruck eine Menge seiner freien Variablen zu.

Nun zu $\mathcal{V}al$. Wir geben das Ergebniss von $\mathcal{V}al\big(\underline{\texttt{exp}}\big) = \rho...$ an, also der Variablenbelegung in einer Umgebung ρ.

$$
\begin{aligned}
&\mathcal{V}al[\![\underline{\texttt{var}}]\!]\rho && = \rho(\underline{\texttt{var}}) \\
&\mathcal{V}al\big[\!\big[\big(\underline{\texttt{exp}}\big)\big]\!\big] && = \mathcal{V}al[\![\underline{\texttt{exp}}]\!] \\
&\mathcal{V}al[\![\underline{\texttt{constr}}_0]\!]\rho && = \rho\big(\underline{\texttt{constr}}_0\big) \text{ z.B. } z \in \mathbb{Z}, b \in \mathbb{B}, \text{etc} \\
&\mathcal{V}al[\![\underline{\texttt{constr}}_n]\!]\rho(n > 0) && = \rho\big(\underline{\texttt{constr}}_n(d_1 ..., d_n)\big) \\
&\mathcal{V}al\big[\!\big[\big(\underline{\texttt{exp}}_1, ..., \underline{\texttt{exp}}_n\big)\big]\!\big]\rho && = \Big(\mathcal{V}al[\![\underline{\texttt{exp}}_1]\!]\rho, ..., \mathcal{V}al[\![\underline{\texttt{exp}}_n]\!]\rho\Big) \\
&\mathcal{V}al\big[\!\big[\big(\underline{\texttt{exp}}_1 \underline{\texttt{exp}}_2\big)\big]\!\big] && = f\Big(\mathcal{V}al[\![\underline{\texttt{exp}}_2]\!]\Big), \text{da } \underline{\texttt{exp}}_1 \text{ eine Funktion } f \text{ ist} \\
&\mathcal{V}al[\![\text{if } \underline{\texttt{exp}}_1 \text{ then } \underline{\texttt{exp}}_2 \text{ else } \underline{\texttt{exp}}_3]\!]\rho && = \begin{cases} \mathcal{V}al[\![\underline{\texttt{exp}}_2]\!] & \text{if } \mathcal{V}al[\![\underline{\texttt{exp}}_1]\!] = \text{true} \\ \mathcal{V}al[\![\underline{\texttt{exp}}_3]\!] & \text{if } \mathcal{V}al[\![\underline{\texttt{exp}}_1]\!] = \text{false} \\ \bot & \text{else} \end{cases} \\
&\mathcal{V}al[\![\text{let var} = \underline{\texttt{exp}} \text{ in } \underline{\texttt{exp}}']\!]\rho && = \mathcal{V}al[\![\underline{\texttt{exp}}']\!]\big(\rho + \big\{\underline{\texttt{var}}/\mathcal{V}al[\![\underline{\texttt{exp}}]\!]\rho\big\}\big) \\
&\mathcal{V}al[\![\backslash\, \underline{\texttt{var}} \to \underline{\texttt{exp}}]\!] && = f, \text{mit } f(d) = \mathcal{V}al[\![\underline{\texttt{exp}}]\!](\rho + \{\underline{\texttt{var}}/d\})
\end{aligned}
$$

3.4 Reduzierung von *Haskell* zu *Simple Haskell*

Nun müssen wir lediglich für jeden normalen Haskell Ausdruck (mit Pattern-Matching) ein äquivalenten *simple* Haskell Ausdruck (ohne Pattern-Matching) erzeugen, und schwups haben wir das Ziel.

Hierfür werden wir Stück für Stück Teilausdrücke mit von uns vordefinierte Funktionen ersetzten, bis keine Regel mehr anwendbar ist. In der Start Environment ω geben wir also nun vordefinierte Funktionen an w_{tr}

`bot` Simple. `bot` $=\bot$. Also Undefiniertheit.

$\texttt{isa}_{\underline{\texttt{constr}}}(d)$
: `true`, falls $d = (\underline{\texttt{constr}}, d_1, ..., d_n)$.
`false` falls $d = (\underline{\texttt{constr}}', d_1, ..., d_n)$,
$\bot$ sonst.

$\texttt{argof}_{\underline{\texttt{constr}}}(d)$
: $(d_1, ...d_n)$ falls $d = (\underline{\texttt{constr}}, d_1, ..., d_n)$,
$\bot$ sonst

$\texttt{isa}_{n-\texttt{tuple}}(d)$
: `true`, falls $d = (d_1, ..., d_n)$,
$\bot$ sonst.

$\texttt{sel}_{n,i}$
: d_i, falls $d = (d_1, ..., d_n)$
$\bot$ sonst

Und nun die 12 Regeln. Die Regeln sind in keiner definierten Reihenfolge zu machen, einfach wie es passt. Da es ein Panikzettel ist, weisen wir euch an das Skript, welche bei 2.2.11 die Transformation beschreibt.

1. **Pattern Matching zu `case`**
2. **Multi-Lambda zu Lambda**
3. **Lambda zu `case`**
4. **`case` zu `match`**

Nun müssen wir noch alle `match` Klauseln autstauschen:

5. **`match` von Variablen**
6. **`match` vom Joker Pattern**
7. **`match` von Konstruktoren**
8. **`match` von leeren Tupeln**
9. **`match` von nicht-leeren Tupeln**
10. **Aufteilung von Deklarationen**
11. **Vereinen von Deklarationen**
12. ↑ Mit mehreren Variablen

Mittels 2.2.12 haben wir gezeigt, dass diese Regln bei einem Korrekt getypten und synataktisch korrekten Programm immer funktionieren und tatsächlich ein *einfaches* Haskell programm raus kommt.

4 Der Lambda-Kalkül

4.1 Lambda-Terme

Die Menge der Lambda-Terme Λ ist definiert als die kleinste Menge, so dass:
$\mathcal{C} \subseteq \Lambda$ Konstanten
$\mathcal{V} \subseteq \Lambda$ Variablen
$(t_1, t_2) \in \Lambda$**, falls** $t_1, t_2 \in \Lambda$ t_1 wird auf t_2 angewendet.
$\lambda x.t \in \Lambda$**, falls** $x \in \mathcal{V}$ **und** $t \in \Lambda$ Abstraktionen, sie stellen Funktionen dar, die jeden Wert x auf den Wert von t abbilden (vgl. Lambda-Funktionen in Haskell)

Variablen in einem Lambda-Term werden als *frei* bezeichnet, wenn diese an kein darüber stehendes Lambda gebunden sind (vgl. freie Variablen in Haskell).
- $((t_1 t_2) t_3) \Rightarrow (t_1 t_2 t_3)$
- $\lambda x.(xx) \Rightarrow \lambda x.xx$

- $\lambda x.\lambda y.t \Rightarrow \lambda x.y.t$

Substitution

Bei einer Substitution werden alle freien vorkommen von x durch t ersetzt. Wir schreiben also z.B. $x[x/t] = t$ (bei Namenskonflikten wird umbenannt). Die Substitution wird dann wie gewohnt rekursiv (z.B. bei Tupeln) weitergeführt.

4.2 Reduktionsregeln

Es gibt Praktisch 3 *Reduktionsregeln* für die Semantik von Lambda Kalkülen.

Für alle gilt, sie dürfen auf Teiloperationen angewendet werden:
- $t_1 \underset{\alpha}{\to} t_2$, dann $(t_1 r) \underset{\alpha}{\to} (t_2 r), (r t_1) \underset{\alpha}{\to} (r t_2), \lambda y.t_1 \underset{\alpha}{\to} \lambda y.t_2$ für alle $r \in \Lambda, y \in \mathcal{V}$

α Reduktion

$$\lambda x.t \underset{\alpha}{\to} \lambda y.t[x/y]$$

falls $y \notin \text{free}(t)$

β Reduktion

Wir Substituieren einfach x durch r in t.

$$(\lambda x.t) r \underset{\beta}{\to} t[x/r]$$

δ Reduktion

Wenn wir δ Regeln der Form $c t_1 ... t_n \to r$ mit $c \in \mathcal{C}, t_1, ..., t_n, r \in \Lambda$ haben, die
- keine Freien Variablen haben
- Und keine Regel aus δ in ihrer **linken Seite** enthalten.
- Keine Konflikte enthalten d.h. $c t_1 ... t_n \to r$, $c t_1 ... t_n \to r'$

Dann ist die δ Reduktion

$$l \underset{\delta}{\to} r$$

für alle $l \to r \in \delta$

4.3 Übersetzung von einfachem Haskell in Lambda-Terme

Dies geschieht mit der Funktion $\mathcal{L}am : \text{Exp} \to \Lambda$

$$\begin{aligned}
\mathcal{L}am(\underline{\text{var}}) &= \underline{\text{var}} \\
\mathcal{L}am(c) &= c \\
\mathcal{L}am\Big(\Big(\underline{\texttt{exp}}, ..., \underline{\texttt{exp}}_n\Big)\Big) &= \texttt{tuple}_n \mathcal{L}am\Big(\underline{\texttt{exp}}_1 ... \mathcal{L}am\Big(\underline{\texttt{exp}}_n\Big)\Big) \\
\mathcal{L}am\Big(\Big(\underline{\texttt{exp}}\Big)\Big) &= \mathcal{L}am\Big(\underline{\texttt{exp}}\Big) \\
\mathcal{L}am\Big(\Big(\underline{\texttt{exp}}_1 \underline{\texttt{exp}}_2\Big)\Big) &= \Big(\mathcal{L}am\Big(\underline{\texttt{exp}}_1\Big) \mathcal{L}am\Big(\underline{\texttt{exp}}_2\Big)\Big) \\
\mathcal{L}am\Big(\texttt{if } \underline{\texttt{exp}}_1 \texttt{ then } \underline{\texttt{exp}}_2 \texttt{ else } \underline{\texttt{exp}}_3\Big) &= \texttt{if } \mathcal{L}am\Big(\underline{\texttt{exp}}_1\Big) \mathcal{L}am\Big(\underline{\texttt{exp}}_2\Big) \mathcal{L}am\Big(\underline{\texttt{exp}}_3\Big) \\
\mathcal{L}am\Big(\texttt{let } \underline{\text{var}} = \underline{\texttt{exp}} \texttt{ in } \underline{\texttt{exp}}'\Big) &= \mathcal{L}am\Big(\underline{\texttt{exp}}'\Big)\Big[\underline{\text{var}}/\Big(\texttt{fix } \Big(\lambda \underline{\text{var}}.\mathcal{L}am\Big(\underline{\texttt{exp}}\Big)\Big)\Big)\Big] \\
\mathcal{L}am\Big(\backslash \underline{\text{var}} \texttt{ -> } \underline{\texttt{exp}}\Big) &= \lambda \underline{\text{var}}.\mathcal{L}am\Big(\underline{\texttt{exp}}\Big)
\end{aligned}$$

5 Typüberprüfung und -inferenz

Bis jetzt wurde bei der Semantik die Typekorrektheit angenommen und nicht überprüft. Hierfür nehmen wir zunächst eine Typannahme A_0, wobei alle vordefinierten Operationen, bereits den **allgemeinsten Zugrundeliegendem Typen** zugewisen haben (z.B. $A_0(42) = \texttt{Int}, \quad A_0(\texttt{not}) = \texttt{Bool} \to \texttt{Bool}$).

Wir müssen nun also noch alle eigens deklarierten Typen inferieren, durch den *Typinferenzalgorithmus* $\mathcal{W}$. Er bekommt eine Typannahme A und einen zu überprüfenden Term t und gibt als Ausgabe eine substituierte Typannahme A' und den allgemeinsten Typen für den Term τ zurück.

$$\mathcal{W}(A + \{c :: \forall a_1, ..., a_n.\tau\}, c) = (\mathrm{id}, \tau[a_1/b_1, ..., a_n/b_n])$$

Das heißt, wenn wir eine Konstante c haben, für die wir bereits den Typen kennen, dann nehmen wir einfach die Identitätsfunktion als Map (da wir sie ja schon haben). Tauschen also praktisch nicht. $b, b_1, ..., b_n$ sind einfach neue Variablen.

Beispiele:
- $\mathcal{W}(A_0 + \{x :: c\}, x)) = (\mathrm{id}, c)$
- $\mathcal{W}(A_0, \mathrm{not}) = (\mathrm{id}, \mathrm{Bool} \to \mathrm{Bool})$ (da not in Haskell vordefiniert ist)

—

Soweit so gut. Nun zu Lambda Funktionen

1. $$\mathcal{W}(A, \lambda x.t) = (\theta, b\theta \to \tau)$$
   $$\text{wobei } \mathcal{W}(A + \{x :: b\}, t) = (\theta, \tau)$$

   Wir finden für $\lambda x.t$ zuerst den Typ von t heraus und nehmen dabei an, dass x den Typ b hat (eine Neue Variable). Hieraus erhalten wir eine Substitution, in der b aber noch offen ist, also $b\theta \to \tau$.

   Beispiel:
   - $\mathcal{W}(\{y :: c\}, \lambda x.y)$ Zuerst finden wir den Typ von y heraus mit zusätzlicher Typannahme $\{x :: d\}$.
2. Lambdaanwendungen
   $$\mathcal{W}(A, (t_1 t_2)) = (\theta_1\theta_2\theta_3, b\theta_3)$$
   $$\text{wobei } \mathcal{W}(A, t_1) = (\theta_1, \tau_1)$$
   $$\text{und } \mathcal{W}(A, t_2) = (\theta_2, \tau_2)$$
   $$\text{und } \theta_3 = \mathrm{mgu}(\tau_1\theta_2, \tau_2 \to b)$$

Bei Funktionsanwendungen $t_1 t_2$ berechnen wir zuerst den Typen von t_1 und dann mit der Substitution θ_1 berechnen wir den Typen von t_2. Somit erhalten wir θ_2 und zusammen also τ_1 und τ_2. Zusammen bilden $\tau_1\theta_2$ und $\tau_2 \to b$, θ_3, also die kombinierte Substitution $(\theta_1\theta_2\theta_3, b\theta_3)$.