Funktionales Programmieren

Carl Philipp Reh

Universität Siegen

24. November 2023

Übersicht

Ziele der Vorlesung:

- Einführung in Haskell.
- Wichtige Typklassen von Haskell.
- Denotationelle Semantik für (eine Teilsprache von) Haskell.
- Typüberprüfung.

Literatur:

- Funktionale Programmierung, Jürgen Giesl, `https://verify.rwth-aachen.de/fp19/FP19.pdf`

Länge einer Liste in C

```
struct element {
  struct data value;
  struct element *next;
};
struct list { struct element *head; };

size_t length(struct list *l) {
  size_t r = 0;
  struct element *cur = l->head;
  while(cur != NULL) {
    cur = cur->next;
    ++r;
  }
  return r;
}
```

Länge einer Liste in Haskell

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

Länge einer Liste in Haskell

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

Die erste Zeile ist eine *Typdeklaration* und sagt, dass `len` eine Funktion ist, die ein `[a]` als Argument bekommt und einen `Int` als Ergebnis liefert.

`[a]` bedeutet „Liste vom Typ `a`“, wobei `a` eine Typvariable ist, die jeden Typ annehmen kann, zum Beispiel `a=Int` oder `a=Bool`.

Länge einer Liste in Haskell

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

Die Definition von Listen besteht aus zwei Fällen:

- `[]` ist die leere Liste, die jeden Typ annehmen kann, also `[] :: [t]` für jeden Typ `t`.
- Wenn `x :: t` und `xs :: [t]`, dann ist `(x : xs) :: [t]` die Liste, die man erhält, wenn man `x` vorne an `xs` hängt.

Zum Beispiel gilt `(5 : (3 : [])) :: [Int]`, was die Liste $(5, 3)$ ist.

Länge einer Liste in Haskell

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

Die Definition von `len` muss die beiden Fälle (leere und nicht leere Liste) unterscheiden. Das geschieht mit **`case`**.

- Wenn die Liste leer ist, geben wir `0` zurück.
- Wenn die Liste nicht leer ist, dann geben wir `1 + len xs` zurück.

Länge einer Liste in Haskell

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

Beispielauswertungen:

```
len [] = 0
```

```
len ("x" : ("y" : []))
  = 1 + (len ("y" : []))
  = 1 + (1 + len [])
  = 1 + (1 + 0)
  = 2
```

Länge einer Liste in Haskell

Weitere Bemerkungen:

- `+` ist in Haskell vordefiniert und ist hier die Addition auf **`Int`**.
- **`Int`** ist ein Typ mit positiven und negativen Zahlen, normalerweise 64 Bit groß.
- `"x"` ist ein String mit dem einzigen Symbol `'x'`.
- Listen können nur aus Elementen eines Typs bestehen. Beispielsweise ist `3 : ("x" : [])` ein Typfehler.
- Statt `(x_1 : ...(x_n : []) ...)` kann man auch `[x_1,...,x_n]` schreiben. Zum Beispiel ist `[1,2] = 1 : (2 : [])`.
- Haskell ist „whitespace-sensitiv“. D.h., die Einrückung in der Definition von `len` ist wichtig.

Länge einer Liste in Haskell

Fundamentale Unterschiede zwischen dem Haskell- und dem C-Programm:

- Keine Schleifen, nur Rekursion.
- Polymorphes Typsystem: Code funktioniert mit jeder Liste.
- Keine Seiteneffekte: Wir verändern nicht den Inhalt von Variablen.
- Automatische Speicherverwaltung: In dem C-Programm müsste man Werte vom Typ `element` mit `malloc` erzeugen und mit `free` freigeben. In Haskell werden die Listen, die wir verwenden, automatisch freigegeben.

Funktionen mit mehreren Argumenten

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

ist eine Funktion, die zwei Argumente erhält. Bei mehreren `->` sind Klammern optional, da implizit rechts geklammert wird. Das heißt, man könnte auch schreiben `add :: Int -> (Int -> Int)`. Dies bedeutet, dass `add` eine Funktion ist, die einen `Int` als Argument erhält und eine *Funktion liefert*, die noch einen `Int` als Argument erhält.

Funktionen mit mehreren Argumenten

Zum Beispiel kann man das Addieren von `5` definieren als

```
add5 :: Int -> Int
add5 = add 5
```

Auch beim Aufrufen von Funktionen werden in der Regel Klammern weggelassen. So bedeutet `add 1 2` dasselbe wie `(add 1) 2`.
Zum Beispiel gilt

```
add5 10 = (add 5) 10 =  add 5 10 = 5 + 10.
```

Funktionen höherer Ordnung

Eine Funktion, die eine andere Funktion als Argument erhält, nennt man *Funktion höherer Ordnung*. Einfaches Beispiel:

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

Funktionen nennt man daher auch „first-class“ in Haskell. D.h., sie können wie normale Daten auch an Funktionen übergeben werden. Beispiel:

```
add10 :: Int -> Int
add10 = twice add5
```

Dann erhalten wir zum Beispiel

```
add10 x = twice add5 x = add5 (add5 x).
```

Lambda-Funktionen

Betrachten wir folgende Funktion `filter`, wobei `filter f l` genau die Elemente `x` von `l` behält, für die `f x == True` gilt:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Man kann dann die geraden Zahlen filtern über

```
filterEven :: [Int] -> [Int]
filterEven = filter (\x -> mod x 2 == 0)
```

Hier ist `\x -> mod x 2 == 0` eine *Lambda-Funktion*, was eine Funktion ohne Namen ist. Diese ist gleichbedeutend mit der benannten Funktion

```
even x = mod x 2 == 0
```

Lambda-Funktionen können Code übersichtlicher machen.

Let-Ausdrücke

Die Syntax `let x = y in z` bindet den Wert `y` an den Namen `x`, welcher in `z` benutzt werden kann. Betrachten wir eine Implementierung von **`filter`**:

```
filter f l = case l of
  [] -> []
  (x : xs) -> if f x
              then (x : filter f xs)
              else filter f xs
```

Wir sehen, dass der Teilausdruck `filter f xs` mehrmals vorkommt. Dies können wir vermeiden, indem wir schreiben

```
  (x : xs) -> let rest = filter f xs
              in if f x then (x : rest)
                 else rest
```

Data-Deklarationen

Data-Deklarationen werden benutzt, um „eigene“ Typen zu definieren. Einer der einfachsten Typen ist `Bool`:

```
data Bool = True | False
```

Dies definiert:

- Dass `Bool` ein Typ ist. Dies nennt man auch einen *Typkonstruktor*.
- Dass `True` und `False` Werte vom Typ `Bool` sind. Diese nennt man auch *Wertkonstruktoren*.

Die `and`-Funktion kann man definieren über

```
and :: Bool -> Bool -> Bool
and x y = case x of
            False -> False
            True -> y
```

Rekursive Data-Deklarationen

Rekursive Data-Deklarationen sind solche, die in ihrer Definition auf sich selbst verweisen. Zum Beispiel kann man die natürlichen Zahlen wie folgt definieren:

```
data Nat = Zero | Succ Nat
```

Dies definiert:

- Den Typkonstruktor `Nat`.
- Den Wertkonstruktor `Zero` vom Typ `Nat`.
- Den Wertkonstruktor `Succ` vom Typ `Nat -> Nat`.

Zum Beispiel steht `Zero` für `0` und `Succ (Succ Zero)` für `2`. Die Vorgängerfunktion kann man definieren über

```
pred :: Nat -> Nat
pred x = case x of
  Zero -> Zero
  Succ y -> y
```

Data-Deklarationen mit Parametern

Data-Deklarationen können auch Typparameter erhalten. Beispiel: Um einen Wert „optional“ zu machen, bietet Haskell den `Maybe`-Typ:

```
data Maybe a = Nothing | Just a
```

Dies definiert:

- Den Typkonstruktor `Maybe`. Für jeden Typ `t` ist `Maybe t` auch ein Typ.
- Den Wert-Konstruktor `Nothing` vom Typ `Maybe a`.
- Den Wert-Konstruktor `Just` vom Typ `a -> Maybe a`.

Beispiel:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y = if y == 0 then Nothing
              else Just (div x y)
```

Eigene Definition von Listen

Listen sind in Haskell vordefiniert und verwenden spezielle Syntax. Wir können aber auch unsere eigenen Listen definieren:

```
data List a = Nil | Cons a (List a)
```

Dies definiert:

- Den Typkonstruktor `List`. Für jeden Typ `t` ist `List t` auch ein Typ.
- Den Wertkonstruktor `Nil` vom Typ `List a`.
- Den Wertkonstruktor `Cons` vom Typ `a -> List a -> List a`.

Listen in Haskell verwenden `[a]` statt `List a` sowie `x : y` statt `Cons x y` und `[]` statt `Nil`.

Kinds

Typen erhalten ebenfalls „Typen“, welche *Kinds* heißen.
Alle „normalen“ Typen wie `Int`, `Int -> Int`, usw. bekommen Kind `*`, was wir als `Int :: *` und `Int -> Int :: *` schreiben.
Typkonstruktoren, die noch einen Typ erwarten, bekommen Kind `* -> *`. Zum Beispiel gilt `List :: * -> *`. Ein Typkonstruktor, der zwei Typen erwartet, hat Kind `* -> * -> *`, usw.
Es gibt auch „Kinds höherer Ordnung“, zum Beispiel `Fix :: (* -> *) -> *`, das wir möglicherweise später kennenlernen werden.

Typklassen

Eine Typklasse legt eine Menge von Funktionen fest, die von einem Typ implementiert werden können. Zum Beispiel wird die Typklasse `Show` benutzt, um einen Typ „auszudrucken“:

```
class Show a where
  show :: a -> String
```

Um die Typklasse `Show` für den Typ `t` verwenden zu können, muss man `Show t =>` an den Funktionstyp schreiben, zum Beispiel:

```
print :: Show a => a -> String
print x = "The␣value␣is␣" ++ show x
```

`++` ist hier Stringkonkatenation.
Eine mögliche Implementierung für `Bool` sähe wie folgt aus:

```
instance Show Bool where
  show x = if x then "True" else "False"
```

Typklassen

Man kann sich die Implementierung von

```
print :: Show a => a -> String
print x = "The␣value␣is␣" ++ show x
```

folgendermaßen vorstellen:

```
print :: (a -> String) -> a -> String
print show x = "The␣value␣is␣" ++ show x
```

Wenn `print` mit `a = Bool` aufgerufen wird, wird das `show` von der `Bool`-Instanz an `print` übergeben.

Vermeiden expliziter Rekursion

Funktionen auf rekursiven Daten wie Listen sind oft selbst rekursiv implementiert. Die Definition von diesen kann man meist durch Hilfsfunktionen, die Rekursion kapseln, vereinfachen. Ein Beispiel hierfür ist `foldr` für Listen:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v l = case l of
  [] -> v
  (x : xs) -> f x (foldr f v xs)
```

Beispielsweise kann man alle Werte einer Liste aufsummieren über

```
sum :: [Int] -> Int
sum l = foldr (+) 0 l
```

Hier muss man den Operator + in Klammern schreiben, um die Funktion `(+) :: Int -> Int -> Int` zu erhalten.

Vermeiden expliziter Rekursion

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v l = case l of
  [] -> v
  (x : xs) -> f x (foldr f v xs)

sum :: [Int] -> Int
sum l = foldr (+) 0 l
```

Als Beispiel erhalten wir

```
sum (3 : (5 : []))
  = foldr (+) 0 (3 : (5 : []))
  = (+) 3 (foldr (+) 0 (5 : []))
  = (+) 3 ((+) 5 (foldr 0 []))
  = (+) 3 ((+) 5 0)
  = 8
```

Functor

Die Funktion `map :: (a -> b) -> [a] -> [b]` bekommt eine Funktion `f :: a -> b` und wendet diese auf alle Elemente einer Liste an. Zum Beispiel gilt

```
map show [1,2] = ["1","2"]
map (\x -> mod x 2 == 0) [1,2,3]
   = [False,True,False]
```

Dies funktioniert nicht bloß auf Listen, sondern auf vielen verschiedenen Datenstrukturen, die Elemente „enthalten“. Hierfür hat Haskell eine eigene Typklasse:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Die Implementierung für Listen ist:

```
instance Functor [] where
  fmap = map
```

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Da hier `f a` und `f b` gebildet werden, muss `f :: * -> *` gelten, was zum Beispiel passend ist für `[] :: * -> *`.
Ein Functor ist also ein Typkonstruktor mit Kind `* -> *`, d.h. eine Abbildung von Typen auf Typen, und eine Funktion, die Funktionen vom Typ `a -> b` auf Funktionen vom Typ `f a -> f b` abbildet.

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Beispielsweise sieht die Implementierung für `Maybe` folgendermaßen aus:

```
instance Functor Maybe where
  fmap f m = case m of
               Nothing -> Nothing
               Just x -> Just (f x)
```

Zum Beispiel gilt

```
fmap (\x -> x + 1) Nothing = Nothing
fmap (\x -> x + 1) (Just 3) = Just 4
```

Functor

Anwendungsbeispiel: Wir wollen ausrechnen, ob ein Benutzer volljährig ist. Dieser gibt sein Alter als String ein, den wir erst umwandeln müssen mit der Funktion

```
stringToInt :: String -> Maybe Int
```

Diese gibt **Nothing** zurück, wenn der String kein gültiger **Int** ist. Zum Testen, ob jemand mindestens 18 ist, definieren wir

```
isAdult :: Int -> Bool
isAdult x  = x >= 18
```

Dann gilt zum Beispiel

```
fmap isAdult (stringToInt "abc") = Nothing
fmap isAdult (stringToInt "10") = Just False
fmap isAdult (stringToInt "20") = Just True
```

Applicative

`Applicative` ist eine Erweiterung von **`Functor`**, die es erlaubt, über mehr als einen Wert auf einmal zu „mappen“. Als Beispiel wollen wir ausgeben, ob ein Benutzer älter ist als der andere. Dazu definieren wir

```
isOlderThan :: Int -> Int -> Bool
```

Mit `fmap` auf **`Maybe`** erhalten wir

```
fmap isOlderThan
  :: Maybe Int -> Maybe (Int -> Bool)
```

Um damit weiter zu arbeiten, brauchen wir eine Funktion

```
Maybe (Int -> Bool) -> Maybe Int
                    -> Maybe Bool
```

die uns von `Applicative` bereitgestellt wird:

```
(<*>) :: f (a -> b) -> f a -> f b
```

Applicative

Die volle Definition von `Applicative` ist

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Die Funktion `pure` „bettet einen Wert ein“.
Die Implementierung für **`Maybe`** sieht folgendermaßen aus

```
instance Applicative Maybe where
  pure = Just
  f <*> m = case f of
    Nothing -> Nothing
    Just f' -> case m of
      Nothing -> Nothing
      Just m' -> Just (f' m')
```

Es gilt also für `g :: a -> b` und `x :: a`, dass

```
Just g <*> Just x = Just (g x)
```

Applicative

Wir können in unserem Beispiel dann schreiben:

```
compareAges ::
  String -> String -> Maybe Bool
compareAges x y =
  (fmap isOlderThan) (stringToInt x)
  <*> (stringToInt y)
```

Um diesen Code besser zu verstehen, ist es wichtig, sich die Typen, die auftreten, anzuschauen: Zunächst haben wir, dass

```
fmap isOlderThan ::
  Maybe Int -> Maybe (Int -> Bool)
```

Dieses wenden wir dann auf `stringToInt x :: Maybe Int` an und erhalten `Maybe (Int -> Bool)`. Als Nächstes übergeben wir dies zusammen mit `stringToInt y :: Maybe Int` an `<*>` und erhalten `Maybe Bool`.

Applicative

`Applicative` erlaubt es, statt `<*>` die Funktion `liftA2` zu implementieren:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Als Beispiel betrachten wir wieder **`Maybe`**:

```
instance Applicative Maybe where
  liftA2 f m m' = case m of
    Nothing -> Nothing
    Just x -> case m' of
      Nothing -> Nothing
      Just x' -> Just (f x x')
```

Wir können dann auch schreiben

```
compareAges x y =
  liftA2 isOlderThan
    (stringToInt x) (stringToInt y)
```

Applicative

Als Anwendungsbeispiel von `Applicative` betrachten wir die Funktion `sequenceA` auf Listen:

```
sequenceA :: Applicative f => [f a] -> f [a]
```

Mit **`Maybe`** ergibt sich

```
sequenceA :: [Maybe a] -> Maybe [a]
```

Hierbei gilt, dass `sequenceA l = Nothing` genau dann, wenn mindestens ein Element in `l` **`Nothing`** ist.
Beispielsweise gilt

```
sequenceA [Nothing, Just 1] = Nothing
sequenceA [Just 1, Just 2] = Just [1,2]
```

Wichtig: Die Implementierung von `sequenceA` auf Listen funktioniert mit jedem `Applicative`.

Monad

Monad ist eine Erweiterung von Applicative, die es erlaubt, „sequentielle Abhängigkeit“ auszudrücken.
Als Beispiel wollen wir die Fehlerbehandlung beim Einlesen eines Alters erweitern. Wir wollen **Int** durch Nat ersetzen, sodass das Alter nicht mehr negativ sein kann. Zum Konvertieren von **Int** zu Nat verwenden wir die Funktion

```
intToNat :: Int -> Maybe Nat
```

Die Funktion stringToInt liefert uns **Maybe Int**. Dieses wollen wir nur an intToNat übergeben, wenn es nicht **Nothing** ist. Wir brauchen also eine Funktion

```
Maybe Int -> (Int -> Maybe Nat) -> Maybe Nat
```

Diese wird uns von **Monad** bereitgestellt:

```
(>>=) :: f a -> (a -> f b) -> f b
```

Monad

Die allgemeine Definition von **Monad** ist

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
```

Zum Beispiel sieht die Implementierung von **Maybe** so aus:

```
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x -> f x
```

Die Funktion f kann nur einen Wert produzieren, wenn m schon nicht **Nothing** ist. Diese „sequentielle Abhängigkeit“ kann man nicht mit Applicative alleine ausdrücken. Zum Beispiel gilt:

```
stringToInt "x" >>= intToNat = Nothing
stringToInt "-5" >>= intToNat = Nothing
stringToInt "2" >>= intToNat
  = Just (Succ (Succ Zero))
```

Monad

Als Anwendungsbeispiel von `Monad` betrachten wir die Funktion

```
join :: Monad f => f (f a) -> f a
```

Diese „entfernt“ eine Schicht eines Typkonstruktors. Zum Beispiel erhalten wir für `Maybe` und `[]`, dass

```
join (Just (Just 10)) = Just 10
join (Just Nothing) = Nothing
join [[]] = []
join [[1,2],[],[3,4],[5]] = [1,2,3,4,5]
```

Auch hier gilt natürlich, dass `join` mit jeder Monade funktioniert.

Monad-Do

```
(>>=) :: m a -> (a -> m b) -> m b
```

Die **do**-Notation erlaubt uns, Ketten aus >>= untereinander zu schreiben. Zum Beispiel betrachten wir

```
divBy :: Int -> Int -> Maybe Int
toNat :: Int -> Maybe Nat
```

Mit >>= könnte man schreiben

```
i :: Maybe Nat
i = stringToInt "5" >>= (\x -> divBy 10 x)
    >>= toNat
```

Die äquivalente **do**-Notation sieht folgendermaßen aus:

```
i = do
    x <- stringToInt "5"
    y <- divBy 10 x
    toNat y
```

Foldable

Wir haben bereits gesehen, wie man `foldr` auf Listen definiert. Andere Datenstrukturen, wie zum Beispiel Bäume, können auch gefaltet werden, wofür Haskell eine Typklasse bereitstellt:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

Das einfachste Beispiel ist wieder `Maybe`:

```
instance Foldable Maybe where
  foldr f s m = case m of
    Nothing -> s
    Just y -> f y s
```

`Maybe` kann man sich also als einen Container vorstellen, der 0 oder 1 Element enthält.

Foldable

Haskell stellt viele Funktionen auf `Foldable` bereit. Ein Beispiel hierfür ist

```
toList :: Foldable t => t a -> [a]
```

Es gilt zum Beispiel

```
toList Nothing = []
toList (Just 1) = [1]
toList [1,2] = [1,2]
```

Man kann hieran erkennen, dass eine Implementierung von **`foldr`** den Elementen eine „Reihenfolge“ gibt. Ein weiteres Beispiel ist

```
sum :: Foldable t -> t Int -> Int
```

Zum Beispiel gilt

```
sum Nothing = 0
sum (Just 1) = 1
```

Monoid

Bei `Foldable` kann man alternativ zu **`foldr`** auch

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

definieren. Dazu müssen wir uns zunächst die Typklasse `Monoid` anschauen. Diese ist eine Erweiterung von `Semigroup`:

```
class Semigroup m where
  (<>) :: m -> m -> m
class Semigroup m => Monoid m where
  mempty :: m
```

Hierbei ist `mempty` das neutrale Element und <> die Verknüpfung. Wie bei Monoiden in der Mathematik soll <> assoziativ sein.

Monoid

Ein `Monoid`, das **`Int`** multipliziert, könnte man zum Beispiel folgendermaßen implementieren:

```
data Times = Mul Int
```

Wir definieren hier zunächst einen neuen Typ `Times`, der dieselben Werte wie **`Int`** enthält. Dieser dient nur dazu, die Monoid-Operationen festzulegen. Die Implementierung ist dann

```
instance Semigroup Times where
  (<>) x y = case x of
               Mul x' -> case y of
                 Mul y' -> Mul (x' * y')
instance Monoid Times where
  mempty = Mul 1
```

Es gilt zum Beispiel `(Mul 2) <> (Mul 5) = Mul 10`.

Foldable

Zurück zu `foldMap`:

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

Die Idee bei `foldMap f l` ist, dass man jedem Element `x :: a` aus `l` über `f` ein Monoid-Element zuordnet. Man startet mit `mempty` und verknüpft alle Monoid-Elemente mit <>.
Für **`Maybe`** kann man `foldMap` folgendermaßen implementieren

```
instance Foldable Maybe where
  foldMap f m = case m of
    Nothing -> mempty
    Just y -> f y
```

Es gilt zum Beispiel

```
foldMap Mul Nothing = Mul 1
foldMap Mul (Just 2) = Mul 2
foldMap Mul [1,2,3] = Mul 6
```

Weitere Typklassen

Es gibt in Haskell sehr viel mehr nützliche Typklassen, die wir uns aber nicht alle anschauen können. Eine nicht vollständige Liste ist

- `Eq` zum Vergleichen.
- `Ord` zum Sortieren.
- `Num` für numerische Operationen wie `+` und `*`.
- `Fractional` für numerische Operationen wie `/`.
- `Traversable` zum „Herausziehen“ von `Applicative`.

Seiteneffekte

Alle bisher betrachteten Funktionen enthalten keine „Seiteneffekte“. Eine Funktion hat einen Seiteneffekt, wenn das Ausführen von ihr irgendeinen Einfluss auf den Rest des Programms hat. Man könnte sich zum Beispiel vorstellen, dass eine Funktion in C intern eine globale Variable schreibt, die dann woanders im Programm gelesen wird.

```
int x = 0;
int f()
{
    ++x;
    return 0;
}
int g()
{
    return x;
}
```

Monad-IO

Auch wenn das vorhin betrachtete Programm „schlechter Code“ ist, gibt es auch „legitime“ Seiteneffekte wie das Schreiben in eine Datei oder das Lesen aus der Standardeingabe.
In Haskell werden solche Seiteneffekte über die `IO`-Monade implementiert. Zum Beispiel gibt es zum Lesen einer Zeile aus der Standardeingabe

```
getLine :: IO String
```

Man kann sich `getLine` als einen Wert vorstellen, der intern beschreibt, wie man durch Ausführen von Seiteneffekten einen Wert vom Typ `String` erhält.
Ein weiteres Beispiel ist

```
putStrLn :: String -> IO ()
```

womit man einen String ausgibt. Das Ergebnis ist vom Typ `()`, was das leere Tupel ist. Das heißt, dass Ausdrucken „nichts“ liefert.

Monad-IO

Es ist nicht möglich, aus einem Wert vom Typ `IO String` einen String zu erhalten. Weil `IO` aber eine Monade ist, kann man `>>=` aufrufen, um an diesen String zu gelangen. Wegen

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

muss man aber einen weiteren Wert vom Typ `IO` produzieren. Das heißt, man kann die `IO`-Monade nie „verlassen“.
Als Beispiel betrachten wir

```
r :: IO ()
r = getLine >>=
    (\x -> putStrLn ("The␣input␣is␣" ++ x))
```

Haskell bietet die `main`-Funktion, welche den `IO`-Wert wirklich „ausführt“, wenn das Programm gestartet wird:

```
main :: IO ()
main = r
```

Auswertung

Wir betrachten folgende Beispielfunktion

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

In welcher Reihenfolge wird folgender Ausdruck ausgewertet?

```
square (2 + 3)
```

Option 1 (Call by Value): Werte *zuerst* den Parameter `2 + 3` aus und ersetze dann die Definition von `square`:

```
square (2 + 3) = square 5 = 5 * 5 = 25
```

Option 2 (Call by Name): Ersetze *zuerst* die Definition von `square` und werte den Parameter aus, wenn nötig:

```
square (2 + 3) = (2 + 3) * (2 + 3)
               = 5 * (2 + 3) = 5 * 5 = 25
```

Auswertung

Nachteil bei Call by Name: Wir werten `2 + 3` zweimal aus.
Vorteil: Man kann sich möglicherweise die Auswertung des Parameters ganz sparen.

Haskell verwendet eine Variation von Call by Name, die Call by Need oder Lazy Evaluation genannt wird. Bei dieser werden alle Teilausdrücke, die aus demselben Ausdruck „entstanden“ sind, nur einmal ausgewertet:

```
square (2 + 3) = (2 + 3) * (2 + 3)
               = 5 * 5 = 25
```

Hier werden beide Teilausdrücke `2 + 3` „auf einmal“ ausgewertet.

Eine formale Beschreibung dieses „schrittweisen Auswertens“ nennt man auch *operationelle Semantik*. Diese ist vor allem dafür nützlich, einen Compiler zu implementieren, aber weniger nützlich, um über das „Verhalten“ eines Programms zu argumentieren.

Denotationelle Semantik

Bei denotationeller Semantik geht es darum, Ausdrücken mathematische Funktionen zuzuordnen. Beispielsweise würden wir gerne der Funktion `square` die mathematische Funktion $[\![\text{square}]\!] : \mathbb{Z} \to \mathbb{Z}$ mit $[\![\text{square}]\!](x) = x^2$ zuordnen.
Dass `Int` einen endlichen Wertebereich hat, ignorieren wir an der Stelle. Ein anderes Problem ist aber, dass es auch divergierende Ausdrücke in Haskell gibt. Zum Beispiel liefert folgende Funktion nie einen Wert:

```
undefined :: a -> a
undefined x = undefined x
```

Also liefert auch `square (undefined 2)` nie einen Wert.

Undefinierte Argumente

Betrachten wir folgende Definition von **and**:

```
and :: Bool -> Bool -> Bool
and x y = case x of
            False -> False
            True -> y
```

Was passiert bei folgendem Programm?

```
and (undefined True) True
```

Da **case** auf **undefined True** angewandt wird, terminiert das Programm nicht. Im Gegensatz dazu gilt

```
and False (undefined True) = False
```

weil **and** sofort **False** liefert, wenn das erste Argument **False** ist, ohne das zweite Argument auszuwerten.

Semantik von square

Wir benötigen für die Semantik von Funktionen also sowohl für die Parameter, als auch für die Ergebnisse noch den speziellen Wert „undefiniert“, den wir mit $\perp$ (Bottom) bezeichnen. Sei $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$. Für die Semantik von `square` können wir also $[\![\mathrm{square}]\!] : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ nehmen mit

$$[\![\mathrm{square}]\!](x) = \begin{cases} \perp & \text{falls } x = \perp, \\ x^2 & \text{falls } x \neq \perp. \end{cases}$$

Was allerdings denotationelle Semantik aufwändig macht, ist die Tatsache, dass wir Funktionen rekursiv definieren können und dass Haskell Funktionen höherer Ordnung hat.
Wir werden stetige Funktionen benötigen, um rekursiv definierten Funktionen eine Semantik zuordnen zu können, was etwas Vorbereitung braucht.

Partielle Ordnungen

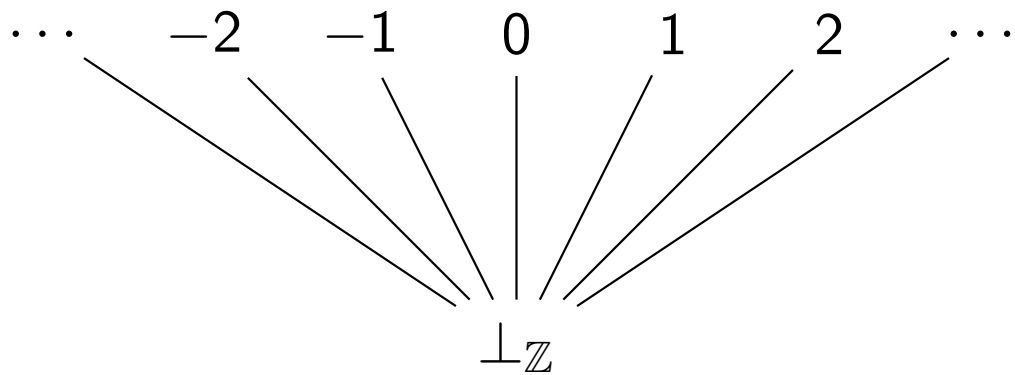
Ein Paar $(D, \sqsubseteq_D)$, wobei D eine Menge ist und $\sqsubseteq_D \subseteq D \times D$, heißt *partielle Ordnung*, wenn $\sqsubseteq_D$ Folgendes erfüllt:

- Reflexivität: Für alle $d \in D$ gilt $d \sqsubseteq_D d$.
- Antisymmetrie: Für alle $d, d' \in D$ gilt: Wenn $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d$, dann gilt $d = d'$. Alternativ: Es gibt keine $d \neq d' \in D$ mit $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d$.
- Transitivität: Für alle $d, d', d'' \in D$ gilt: Wenn $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d''$, dann gilt $d \sqsubseteq_D d''$.

Wir werden oft einfach nur D statt $(D, \sqsubseteq_D)$ und $\sqsubseteq$ statt $\sqsubseteq_D$ schreiben, wenn $\sqsubseteq_D$ aus dem Kontext bekannt ist.
Alle Ordnungen, die wir betrachten, haben die Intuition, dass $d \sqsubseteq_D d'$ gilt, wenn d „weniger definiert“ ist als d'.

Flache Ordnungen

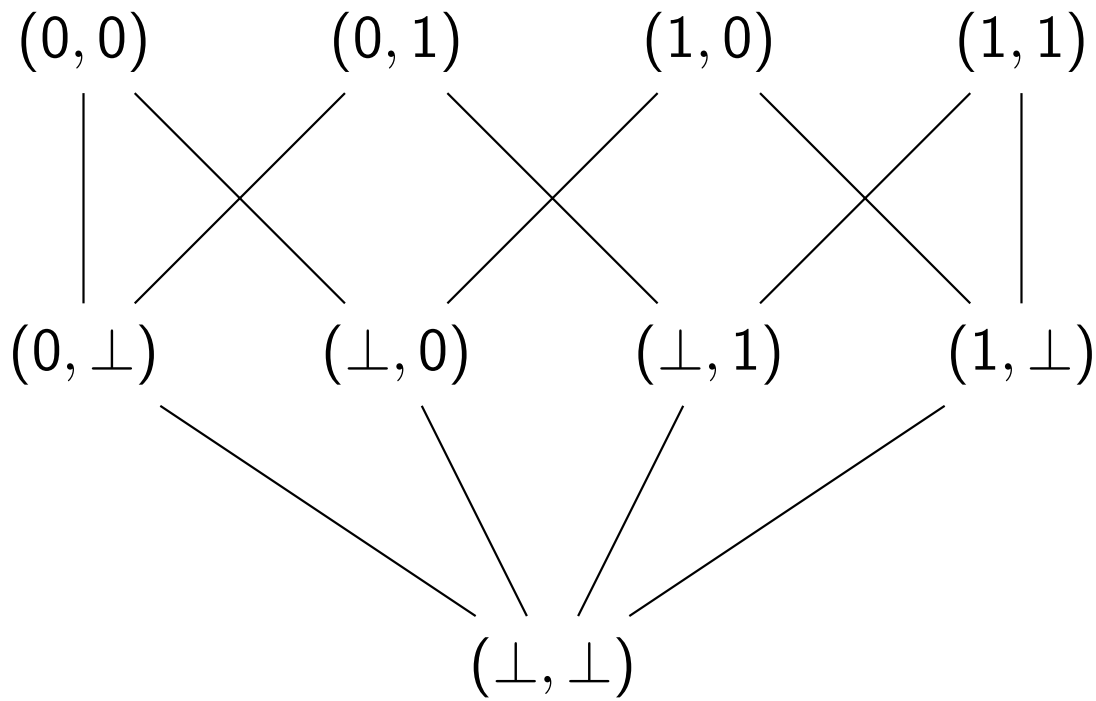
Für eine Menge D ist die *flache Ordnung* $(D_\perp, \sqsubseteq_{D_\perp})$ definiert als $D_\perp = D \uplus \{\perp_D\}$ und $d \sqsubseteq_{D_\perp} d'$ genau dann, wenn $d = \perp_D$ oder $d = d'$. Dies definiert eine partielle Ordnung.
Beispiel: $(\mathbb{Z}_\perp, \sqsubseteq_{\mathbb{Z}_\perp})$ ist die Ordnung, bei der gilt, dass $\perp_\mathbb{Z} \sqsubseteq_{\mathbb{Z}_\perp} d$ für alle $d \in \mathbb{Z}_\perp$. In diesem Sinne ist also $\perp_\mathbb{Z}$ „undefinierter“ als alle ganzen Zahlen, aber keine ganze Zahl ist „undefinierter“ als eine andere. Grafisch:



Wir schreiben oft einfach nur $\perp$ statt $\perp_D$, wenn D aus dem Kontext bekannt ist.

Produkt-Ordnung

Zu $n \geq 0$ partiellen Ordnungen $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ definieren wir die *Produkt-Ordnung* $(D_1 \times \cdots \times D_n, \sqsubseteq_{D_1 \times \cdots \times D_n})$ wie folgt: Es gilt $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \cdots \times D_n} (d'_1, \ldots, d'_n)$ genau dann, wenn $d_i \sqsubseteq_{D_i} d'_i$ für alle $1 \leq i \leq n$.
Als Beispiel sei $\mathbb{Z}_2 = \{0, 1\}$ und wir betrachten die partielle Ordnung $((\mathbb{Z}_2)_\perp \times (\mathbb{Z}_2)_\perp, \sqsubseteq_{(\mathbb{Z}_2)_\perp \times (\mathbb{Z}_2)_\perp})$. Grafisch:



Die Produkt-Ordnung ist eine partielle Ordnung

Lemma 1

Die Produkt-Ordnung ist eine partielle Ordnung.

Beweis.

Reflexivität: Sei $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$. Es gilt $d_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$, weil $\sqsubseteq_{D_i}$ reflexiv ist. Also folgt $(d_1, \ldots, d_n) \sqsubseteq (d_1, \ldots, d_n)$.
Antisymmetrie: Wenn $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ und $(d'_1, \ldots, d'_n) \sqsubseteq (d_1, \ldots, d_n)$, dann gilt $d_i \sqsubseteq_{D_i} d'_i$ und $d'_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$. Weil $\sqsubseteq_{D_i}$ antisymmetrisch ist, gilt $d_i = d'_i$ für alle $1 \leq i \leq n$. Also folgt $(d_1, \ldots, d_n) = (d'_1, \ldots, d'_n)$.
Transitivität: Wenn $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ und $(d'_1, \ldots, d'_n) \sqsubseteq (d''_1, \ldots, d''_n)$, dann gilt $d_i \sqsubseteq_{D_i} d'_i$ und $d'_i \sqsubseteq_{D_i} d''_i$ für alle $1 \leq i \leq n$. Wegen Transitivität von $\sqsubseteq_{D_i}$ gilt auch $d_i \sqsubseteq_{D_i} d''_i$ für alle $1 \leq i \leq n$. Also folgt $(d_1, \ldots, d_n) \sqsubseteq (d''_1, \ldots, d''_n)$. □

Ordnung auf Funktionen

Seien D eine Menge und $(E, \sqsubseteq_E)$ eine partielle Ordnung. Wir definieren die partielle Ordnung $(D \to E, \sqsubseteq_{D \to E})$ wie folgt: Für $f, f' \colon D \to E$ gilt $f \sqsubseteq_{D \to E} f'$ genau dann, wenn für alle $d \in D$ gilt, dass $f(d) \sqsubseteq_E f'(d)$.
In gewissem Sinne ist dies analog zu Tupeln, wenn man sich eine Funktion $D \to E$ als ein $|D|$-stelliges Tupel mit Komponenten aus E vorstellt. $|D|$ kann allerdings unendlich sein.

Lemma 2

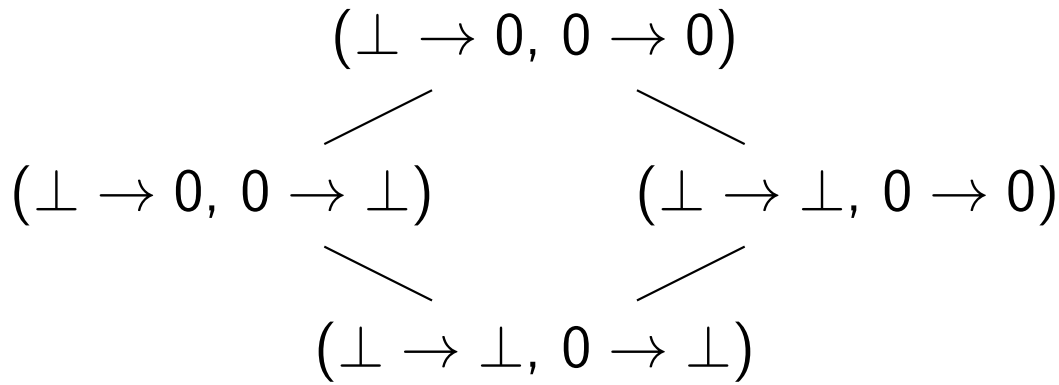
$(D \to E, \sqsubseteq_{D \to E})$ *ist eine partielle Ordnung.*

Beweis.

Übung. □

Ordnung auf Funktionen

Sei $\mathbb{Z}_1 = \{0\}$. Wir betrachten die Funktionen $(\mathbb{Z}_1)_\perp \to (\mathbb{Z}_1)_\perp$, für die wir folgende Ordnung erhalten:



Die Schreibweise $(\perp \to x,\ 0 \to y)$ bedeutet, dass $\perp$ auf x und 0 auf y abgebildet werden.
Die Funktion $(\perp \to \perp,\ 0 \to \perp)$ ist also „am wenigsten definiert“ und die Funktion $(\perp \to 0,\ 0 \to 0)$ ist „am meisten definiert“.

Monotone Funktionen

Seien $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ partielle Ordnungen. Eine Funktion $f : D \to E$ heißt *monoton*, wenn für alle $d, d' \in D$ mit $d \sqsubseteq_D d'$ gilt, dass $f(d) \sqsubseteq_E f(d')$.
Die Intuition ist, dass ein Argument, das „mehr definiert“ ist, nicht zu einem Ergebnis führen kann, das „weniger“ definiert ist.
In unserem vorherigen Beispiel ist die Funktion f mit $f(\bot) = 0$ und $f(0) = \bot$ also nicht monoton, weil $\bot \sqsubseteq 0$ gilt, aber $f(\bot) = 0 \not\sqsubseteq \bot = f(0)$. Diese Funktion lässt sich auch nicht in Haskell implementieren, weil die Implementierung testen müsste, ob ihr Argument divergiert.
Die anderen drei Funktionen sind monoton, da

- Identitäten $f(x) = x$ monoton sind, denn es gilt für alle $d \sqsubseteq d'$, dass $f(d) = d \sqsubseteq d' = f(d')$, und
- konstante Funktionen $f(x) = c$ monoton sind, denn es gilt für alle $d \sqsubseteq d'$, dass $f(d) = c \sqsubseteq c = f(d')$.

Ketten

Seien $(\mathbb{N}, \leq)$ und $(D, \sqsubseteq_D)$ partielle Ordnungen. Eine monotone Funktion $c\colon \mathbb{N} \to D$ heißt *Kette* in $(D, \sqsubseteq_D)$. Wir sagen auch einfach, dass c eine Kette ist, wenn $(D, \sqsubseteq_D)$ aus dem Kontext bekannt ist.
Wegen Monotonie ist das Bild $\mathrm{Img}(c)$ von c also eine abzählbar unendliche Folge $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \ldots$ von Elementen $d_0, d_1, \cdots \in D$, wobei $d_i = c(i)$ für $i \in \mathbb{N}$. Wir sagen, dass die Kette endlich ist, wenn $\mathrm{Img}(c)$ endlich ist. In dem Fall ist $\mathrm{Img}(c) = \{d_0, \ldots, d_n\}$ für $n \geq 0$, wobei $d_0 \sqsubseteq \cdots \sqsubseteq d_n$.

Eine endliche Kette in $\mathbb{Z}_\perp$ ist zum Beispiel $\perp \sqsubseteq 3$.
Endliche Ketten in $\mathbb{Z}_\perp \times \mathbb{Z}_\perp$ sind zum Beispiel
$(\perp, \perp) \sqsubseteq (5, \perp) \sqsubseteq (5, 9)$ und $(\perp, 3) \sqsubseteq (7, 3)$.

Die Intuition bei Ketten ist, dass größere Elemente „mehr definiert“ sind als kleinere.

Ketten

Unendliche Ketten treten bei Funktionen auf. Wir können uns zum Beispiel an die Definition der Fakultät wie folgt annähern: Sei $\mathsf{fact}_i : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ für $i \in \mathbb{N}$ definiert als

$$\mathsf{fact}_0(x) = \perp$$

$$\mathsf{fact}_{i+1}(x) = \begin{cases} \perp & \text{falls } x = \perp \text{ oder } x > i, \\ 1 & \text{falls } x < 0, \\ x! & \text{falls } 0 \leq x \leq i. \end{cases}$$

Dann ist $\mathsf{fact}_0 \sqsubseteq \mathsf{fact}_1 \sqsubseteq \mathsf{fact}_2 \sqsubseteq \cdots$ eine Kette in $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$, wobei fact_{i+1} nur „bis i“ definiert ist. Als Funktion ist dies $c : \mathbb{N} \to (\mathbb{Z}_\perp \to \mathbb{Z}_\perp)$ mit $c(i) = \mathsf{fact}_i$ für $i \in \mathbb{N}$.

Kleinste obere Schranke

Sei $(D, \sqsubseteq_D)$ eine partielle Ordnung und $M \subseteq D$. Ein Element $d \in D$ heißt *obere Schranke* von M, wenn für alle $d' \in M$ gilt, dass $d' \sqsubseteq_D d$. Wir schreiben hierfür $M \sqsubseteq_D d$. Ferner heißt d *kleinste obere Schranke* von M, wenn für alle (anderen) oberen Schranken $M \sqsubseteq_D d''$ gilt, dass $d \sqsubseteq_D d''$. Dieses Element ist eindeutig (falls es existiert) und wir schreiben hierfür $\sqcup M$.
Für eine Kette $c\colon \mathbb{N} \to D$ ist die *kleinste obere Schranke von* c die kleinste obere Schranke von $\mathrm{Img}(c) \subseteq D$ (falls diese existiert). Wir schreiben hierfür $\sqcup c$.

Für die unendliche Kette $c(i) = \mathrm{fact}_i$ ist $\sqcup c$ die Fakultät selbst, also die Funktion $\mathrm{fact}\colon \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ mit

$$\mathrm{fact}(x) = \begin{cases} \perp & \text{falls } x = \perp, \\ 1 & \text{falls } x < 0, \\ x! & \text{falls } x \geq 0. \end{cases}$$

Complete partial order

Sei $(D, \sqsubseteq_D)$ eine partielle Ordnung. Ein Element $d \in D$ heißt *kleinstes Element* bezüglich $\sqsubseteq_D$, wenn für alle $d' \in D$ gilt, dass $d \sqsubseteq_D d'$. Dieses ist eindeutig und wir bezeichnen es mit $\bot_D$ (falls es existiert).
$(D, \sqsubseteq_D)$ heißt *CPO* (complete partial order bzw. vollständige partielle Ordnung), wenn

- für jede Kette $c \colon \mathbb{N} \to D$ die kleinste obere Schranke $\sqcup c$ existiert und
- wenn das kleinste Element $\bot_D$ existiert.

Wir werden im Weiteren zeigen, dass alle partiellen Ordnungen, die wir bisher betrachten haben, tatsächlich CPOs sind. So können wir zum Beispiel sicherstellen, dass $\sqcup c$ für die Kette $c(i) = \text{fact}_i$ existiert. Das Vorhandensein des kleinsten Elements wird später wichtig sein.

Die flache Ordnung ist eine CPO

Zunächst stellen wir fest, dass für jede endliche Kette $c\colon \mathbb{N} \to D$ die kleinste obere Schranke existiert, was das „letzte“ Element der Kette ist (Beweis: Übung). Wir können dann zeigen:

Lemma 3

Die flache Ordnung $(D_\bot, \sqsubseteq_{D_\bot})$ *ist eine CPO.*

Beweis.

$\bot_D$ ist das kleinste Element dieser Ordnung, denn für alle $d \in D_\bot$ gilt, dass $\bot_D \sqsubseteq d$. Sei $c\colon \mathbb{N} \to D$ eine Kette. Es gilt entweder $\mathrm{Img}(c) = \{\bot_D\}$, oder es gibt ein $d \in D$ mit $\mathrm{Img}(c) = \{d\}$ oder $\mathrm{Img}(c) = \{\bot_D, d\}$. Also ist c endlich und damit existiert $\sqcup c$. □

Monotone Funktionen und Ketten

Die *Komposition* von zwei Funktionen $f : D \to E$ und $g : E \to F$ ist die Funktion $g \circ f : D \to F$ mit $(g \circ f)(x) = g(f(x))$.
Um ein paar der Beweise zu vereinfachen, zeigen wir zunächst:

Lemma 4

Die Komposition $g \circ f$ zweier monotoner Funktionen $f : D \to E$ und $g : E \to F$ ist monoton.

Beweis.

Übung. □

Daraus folgt sofort:

Folgerung 5

1. *Wenn $c : \mathbb{N} \to D$ eine Kette ist und $f : D \to E$ monoton ist, dann ist $f \circ c : \mathbb{N} \to E$ eine Kette.*
2. *Wenn $c : \mathbb{N} \to D$ eine Kette ist und $f : \mathbb{N} \to \mathbb{N}$ eine Kette in $(\mathbb{N}, \leq)$ ist, dann ist $c \circ f : \mathbb{N} \to D$ eine Kette.*

Projektion und Applikation

In Kombination mit Folgerung 5 ist es nützlich zu zeigen, dass gewisse Funktionen monoton sind. Wir werden in den folgenden zwei Beweisen die i'te Projektion und die Applikation benötigen: Die *i'te Projektion* $\pi_i \colon D_1 \times \cdots \times D_n \to D_i$ für $1 \leq i \leq n$ ist definiert als $\pi_i(d_1, \ldots, d_n) = d_i$. Die *Applikation von d* $a_d \colon (D \to E) \to E$ für $d \in D$ ist definiert als $a_d(f) = f(d)$.

Lemma 6

Die i'te Projektion und die Applikation sind monoton.

Beweis.

Übung. □

Mit der Vorstellung, dass eine Funktion $D \to E$ ein $|D|$-stelliges Tupel ist, können wir uns auch a_d als die „d'te Projektion“ einer Funktion vorstellen.

Die Produkt-Ordnung ist eine CPO (Teil 1)

Lemma 7

Wenn $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ *CPOs sind, dann ist auch* $(D_1 \times \cdots \times D_n, \sqsubseteq_{D_1 \times \cdots \times D_n})$ *eine CPO.*

Beweis.

$(\bot_{D_1}, \ldots, \bot_{D_n})$ ist das kleinste Element, denn sei $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$. Dann gilt wegen $\bot_{D_i} \sqsubseteq_{D_i} d_i$ für $1 \leq i \leq n$ auch, dass $(\bot_{D_1}, \ldots, \bot_{D_n}) \sqsubseteq (d_1, \ldots, d_n)$.
Sei $c \colon \mathbb{N} \to D_1 \times \cdots \times D_n$ eine Kette. Dann ist $c_i \colon \mathbb{N} \to D_i$ mit $c_i = \pi_i \circ c$ für $1 \leq i \leq n$ eine Kette nach Folgerung 5, da π_i monoton ist nach Lemma 6. Also existiert jeweils $\sqcup c_i$, weil $(D_i, \sqsubseteq_{D_i})$ eine CPO ist.

Die Produkt-Ordnung ist eine CPO (Teil 2)

Beweis.

Sei $s = (\sqcup c_1, \ldots, \sqcup c_n)$. Wir wollen zeigen, dass $\sqcup c = s$.
Um zu zeigen, dass s eine obere Schranke ist, sei $(d_1, \ldots, d_n) \in \mathrm{Img}(c)$. Dann gilt für alle $1 \leq i \leq n$, dass $d_i \in \mathrm{Img}(c_i)$, also $d_i \sqsubseteq \sqcup c_i$, woraus $(d_1, \ldots, d_n) \sqsubseteq s$ folgt.
Um zu zeigen, dass s die kleinste obere Schranke ist, sei $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ mit $\mathrm{Img}(c) \sqsubseteq (d_1, \ldots, d_n)$. Dann gilt $\mathrm{Img}(c_i) \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$, also auch $\sqcup c_i \sqsubseteq_{D_i} d_i$. Daraus folgt, dass $s \sqsubseteq (d_1, \ldots, d_n)$. □

Die Ordnung auf Funktionen ist eine CPO (Teil 1)

Lemma 8

Sei D eine Menge und $(E, \sqsubseteq_E)$ eine CPO. Dann ist $(D \to E, \sqsubseteq_{D \to E})$ eine CPO.

Beweis.

Das kleinste Element ist die Funktion $f\colon D \to E$ mit $f(x) = \bot_E$, denn für alle $g\colon D \to E$ gilt, dass $f(d) = \bot_E \sqsubseteq g(d)$ für alle $d \in D$, und somit $f \sqsubseteq g$.
Sei $c\colon \mathbb{N} \to (D \to E)$ eine Kette. Dann ist $c_d\colon \mathbb{N} \to E$ für $d \in D$ mit $c_d = a_d \circ c$ eine Kette nach Folgerung 5, weil a_d monoton ist nach Lemma 6. Also existiert jeweils $\sqcup c_d$, weil $(E, \sqsubseteq_E)$ eine CPO ist.

Die Ordnung auf Funktionen ist eine CPO (Teil 2)

Beweis.

Sei $f : D \to E$ definiert als $f(d) = \sqcup c_d$. Wir wollen zeigen, dass $\sqcup c = f$.

Um zu zeigen, dass f eine obere Schranke ist, sei $g \in \mathrm{Img}(c)$. Für alle $d \in D$ gilt $g(d) \in \mathrm{Img}(c_d)$, also $g(d) \sqsubseteq_E \sqcup c_d$, woraus $g \sqsubseteq f$ folgt.

Um zu zeigen, dass f die kleinste obere Schranke ist, sei $g : D \to E$ mit $\mathrm{Img}(c) \sqsubseteq g$. Dann gilt $\mathrm{Img}(c_d) \sqsubseteq_E g(d)$ für alle $d \in D$, also auch $\sqcup c_d \sqsubseteq_E g(d)$. Daraus folgt, dass $f \sqsubseteq g$. □

Stetige Funktionen

Seien $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ CPOs. Eine monotone Funktion $f : D \to E$ heißt *stetig*, wenn für jede Kette $c : \mathbb{N} \to D$ gilt, dass

$$f(\sqcup c) = \sqcup(f \circ c).$$

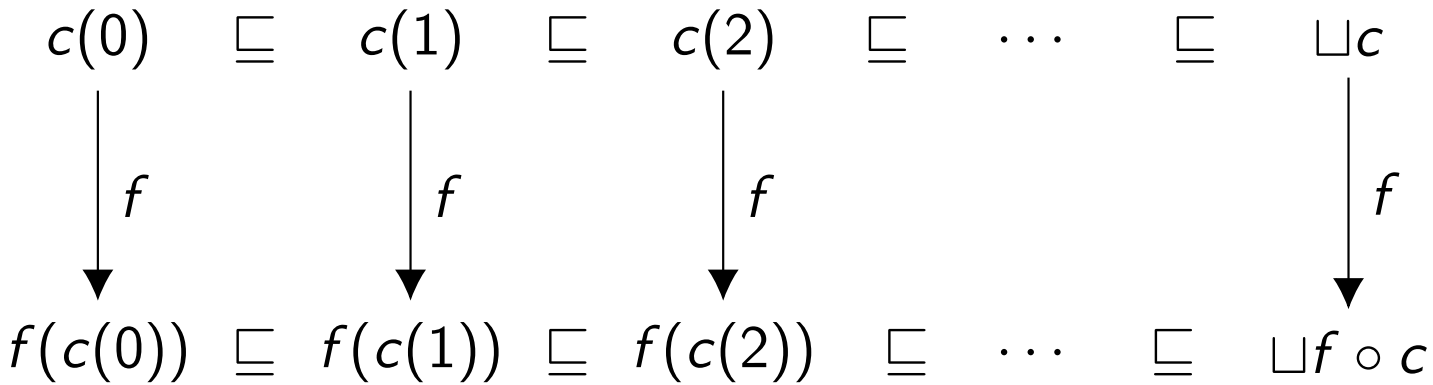
Alle oberen Schranken existieren, weil D und E CPOs sind. Außerdem ist $f \circ c$ eine Kette in E nach Folgerung 5. Wenn es sich anbietet, werden wir λ-Notation für Funktionen benutzen. Statt zu sagen, dass $f : X \to Y$ eine Funktion ist mit $f(x) = y$, schreibt man einfach $\lambda x.y$, wobei X und Y aus dem Kontext ersichtlich sind. Die Stetigkeitsbedingung kann man also auch schreiben als

$$f(\sqcup \lambda i.c(i)) = \sqcup \lambda i.(f(c(i))).$$

Die Menge der stetigen Funktionen bezeichnen wir mit $[(D, \sqsubseteq_D) \to (E, \sqsubseteq_E)]$ oder einfach nur $[D \to E]$, wenn die CPOs aus dem Kontext bekannt sind.

Stetige Funktionen

Die Kette c bildet auf „immer definiertere“ Elemente in D ab. Wenn wir $f \circ c$ bilden, erhalten wir wegen Monotonie von f also immer definiertere Elemente in E. Die kleinste obere Schranke davon muss mit dem Element übereinstimmen, das wir erhalten, wenn wir f auf die kleinste obere Schranke von c anwenden. Man kann also entweder $c(0), c(1), \ldots$ bis $\sqcup c$ verfolgen und dann f anwenden, oder man kann $f(c(0)), f(c(1)), \ldots$ bis $\sqcup(f \circ c)$ verfolgen. Grafisch:



Beispiele für stetige Funktionen

- Die Identität $f : D \to D$ mit $f(x) = x$ ist stetig, da $f(\sqcup c) = \sqcup c = \sqcup(f \circ c)$.
- Jede konstante Funktion $f : D \to E$ mit $f(x) = e$ für $e \in E$ ist stetig, denn $(f \circ c)(x) = e$ für alle $x \in D$ und somit $\sqcup(f \circ c) = e = f(\sqcup c)$.
- Die Projektionen $\pi_i : D_1 \times \cdots \times D_n \to D$ sind stetig. Wegen $\sqcup c = (\sqcup \pi_1 \circ c, \ldots, \sqcup \pi_n \circ c)$ gilt, dass $\pi_i(\sqcup c) = \sqcup(\pi_i \circ c)$.

Doppelt indizierte Ketten

Wenn man zeigen will, dass Funktionen stetig sind, kommt es oft vor, dass man mit mehreren Ketten auf einmal zu tun hat. Sei $(D, \sqsubseteq_D)$ eine partielle Ordnung. Eine Funktion $c\colon \mathbb{N}^2 \to D$ heißt *doppelt indizierte Kette in D*, wenn $\lambda j.c(i,j)$ für jedes $i \in \mathbb{N}$ und $\lambda i.c(i,j)$ für jedes $j \in \mathbb{N}$ Ketten sind.

Lemma 9

Sei $c\colon \mathbb{N}^2 \to D$ eine doppelt indizierte Kette. Dann sind $\lambda i.\sqcup\lambda j.c(i,j)$, $\lambda j.\sqcup\lambda i.c(i,j)$ und $\lambda k.c(k,k)$ Ketten in D und es gilt

$$\sqcup\lambda i.\sqcup\lambda j.c(i,j) = \sqcup\lambda j.\sqcup\lambda i.c(i,j) = \sqcup\lambda k.c(k,k).$$

Doppelt indizierte Ketten

Grafisch:

$$
\begin{array}{ccccccc}
c(0,0) & \sqsubseteq & c(0,1) & \sqsubseteq & \cdots & \sqsubseteq & \sqcup\lambda j.c(0,j) \\
\sqcap| & & \sqcap| & & & & \sqcap| \\
c(1,0) & \sqsubseteq & c(1,1) & \sqsubseteq & \cdots & \sqsubseteq & \sqcup\lambda j.c(1,j) \\
\sqcap| & & \sqcap| & & & & \sqcap| \\
\vdots & & \vdots & & \ddots & & \vdots \\
\sqcap| & & \sqcap| & & & & \sqcap| \\
\sqcup\lambda i.c(i,0) & \sqsubseteq & \sqcup\lambda i.c(i,1) & \sqsubseteq & \cdots & \sqsubseteq & \sqcup\lambda i.\sqcup\lambda j.c(i,j)
\end{array}
$$

Doppelt indizierte Ketten, Beweis (Teil 1)

Beweis.

Seien $i_1, i_2 \in \mathbb{N}$ mit $i_1 \leq i_2$. Um zu zeigen, dass $\lambda i.\sqcup\lambda j.c(i,j)$ eine Kette ist, müssen wir $\sqcup\lambda j.c(i_1,j) \sqsubseteq \sqcup\lambda j.c(i_2,j)$ zeigen. Da $\lambda i.c(i,j_1)$ für alle $j_1 \in \mathbb{N}$ eine Kette ist, gilt $c(i_1,j_1) \sqsubseteq c(i_2,j_1)$. Außerdem ist $\lambda j.c(i_2,j)$ eine Kette. Wegen $c(i_2,j_1) \sqsubseteq \sqcup\lambda j.c(i_2,j)$ folgt, dass $c(i_1,j_1) \sqsubseteq \sqcup\lambda j.c(i_2,j)$ für alle $j_1 \in \mathbb{N}$. Da also $\sqcup\lambda j.c(i_2,j)$ obere Schranke von $\lambda j.c(i_1,j)$ ist, liegt es über der kleinsten oberen Schranke von $\lambda j.c(i_1,j)$, also $\sqcup\lambda j.c(i_1,j) \sqsubseteq \sqcup\lambda j.c(i_2,j)$.
Analog folgt, dass auch $\lambda j.\sqcup\lambda i.c(i,j)$ eine Kette ist.
Außerdem gilt für alle $i_1, i_2 \in \mathbb{N}$ mit $i_1 \leq i_2$, dass $c(i_1,i_1) \sqsubseteq c(i_1,i_2) \sqsubseteq c(i_2,i_2)$, weswegen auch $\lambda k.c(k,k)$ eine Kette ist.

Doppelt indizierte Ketten, Beweis (Teil 2)

Beweis.

Sei $i_1 \in \mathbb{N}$. Dann gilt für alle $i_2 \in \mathbb{N}$ mit $i_1 \leq i_2$, dass $c(i_1, i_2) \sqsubseteq c(i_2, i_2) \sqsubseteq \sqcup\lambda j.c(j, j)$. Die Argumentation im Fall, dass $i_2 \leq i_1$, ist analog. Wir haben also, dass $\sqcup\lambda j.c(j, j)$ obere Schranke von $\lambda j.c(i_1, j)$ ist, also $\sqcup\lambda j.c(i_1, j) \sqsubseteq \sqcup\lambda j.c(j, j)$. Ebenso gilt $c(i_1, i_1) \sqsubseteq \sqcup\lambda j.c(i_1, j)$. Damit erhalten wir

$$c(i_1, i_1) \sqsubseteq \sqcup\lambda j.c(i_1, j) \sqsubseteq \sqcup\lambda j.c(j, j),$$

woraus folgt, dass

$$\sqcup\lambda i.c(i, i) \sqsubseteq \sqcup\lambda i.\sqcup\lambda j.c(i, j) \sqsubseteq \sqcup\lambda i.\sqcup\lambda j.c(j, j) = \sqcup\lambda j.c(j, j).$$

Da $(D, \sqsubseteq_D)$ eine partielle Ordnung ist, erhalten wir $\sqcup\lambda k.c(k, k) = \sqcup\lambda i.\sqcup\lambda j.c(i, j)$. Analog folgt, dass $\sqcup\lambda k.c(k, k) = \sqcup\lambda j.\sqcup\lambda i.c(i, j)$, und somit auch $\sqcup\lambda i.\sqcup\lambda j.c(i, j) = \sqcup\lambda j.\sqcup\lambda i.c(i, j)$. □

Die stetigen Funktionen sind eine CPO (Teil 1)

Wir haben bereits gezeigt, dass $(D \to E, \sqsubseteq_{D \to E})$ eine CPO ist. Wenn wir $D \to E$ auf $[D \to E]$ beschränken wollen, müssen wir noch Folgendes zeigen:

Lemma 10

Wenn $(D, \sqsubseteq_D)$ *und* $(E, \sqsubseteq_E)$ *CPOs sind, dann ist auch* $([D \to E], \sqsubseteq_{D \to E})$ *eine CPO.*

Beweis.

Die Funktion $f : D \to E$ mit $f(x) = \bot_E$ ist wieder das kleinste Element. Diese Funktion ist stetig, weil sie konstant ist.

Die stetigen Funktionen sind eine CPO (Teil 2)

Beweis.

Sei $c : \mathbb{N} \to [D \to E]$ eine Kette. Sei wieder $f : D \to E$ definiert als $f(d) = \sqcup(a_d \circ c)$, also $f(d) = \sqcup\lambda i.c(i)(d)$. Wir wollen zeigen, dass f stetig ist. Sei also $c' : \mathbb{N} \to D$ eine weitere Kette. Dann ist zu zeigen, dass $f(\sqcup c') = \sqcup(f \circ c')$ bzw.

$$f(\sqcup\lambda j.c'(j)) = \sqcup\lambda j.f(c'(j)).$$

Nach Definition von f können wir $f(\sqcup\lambda j.c'(j))$ umformen zu $\sqcup\lambda i.c(i)(\sqcup\lambda j.c'(j))$. Da $c(i)$ stetig ist, ist dies gleich $\sqcup\lambda i.\sqcup\lambda j.(c(i)(c'(j)))$. Dann ist $g : \mathbb{N}^2 \to E$ mit $g(i,j) = c(i)(c'(j))$ eine doppelt indizierte Kette (Beweis: Übung). Nach Lemma 9 können wir also umformen zu $\sqcup\lambda j.\sqcup\lambda i.(c(i)(c'(j)))$. Nach Definition von f ist dies gleich $\sqcup\lambda j.f(c'(j))$. □

Stetigkeit der Applikation

Als weiteres Beispiel einer stetigen Funktion betrachten wir die *Applikationsfunktion* $a\colon [D \to E] \times D \to E$ mit $a(f, x) = f(x)$. Um zu zeigen, dass a stetig ist, sei $c\colon \mathbb{N} \to [D \to E] \times D$ eine Kette mit $c(i) = (f_i, d_i)$. Wegen $\sqcup c = (\pi_1 \circ \sqcup c, \pi_2 \circ \sqcup c)$ können wir schreiben $a(\sqcup c) = (\sqcup \lambda i.f_i)(\sqcup \lambda j.d_j)$. Da $\sqcup \lambda i.f_i$ stetig ist, können wir dies umformen zu $\sqcup \lambda j.\sqcup \lambda i.f_i(d_j)$. Dann ist $g\colon \mathbb{N}^2 \to E$ mit $g(i, j) = f_i(d_j)$ eine doppelt indizierte Kette. Wir können also $\sqcup \lambda j.\sqcup \lambda i.f_i(d_j)$ nach Lemma 9 umformen zu $\sqcup \lambda k.f_k(d_k)$, was gleich $\sqcup(a \circ c)$ ist.

Beispiel einer nicht stetigen Funktion

Ein Beispiel für eine nicht stetige Funktion ist der Totalitätstest. Sei $t\colon (\mathbb{N} \to \mathbb{N}_\perp) \to (\mathbb{Z}_2)_\perp$ definiert als

$$t(f) = \begin{cases} 1 & \text{falls } f(i) \neq \perp \text{ für alle } i \in \mathbb{N}, \\ 0 & \text{sonst.} \end{cases}$$

Für jedes $i \in \mathbb{N}$ sei $f_i\colon \mathbb{N} \to \mathbb{N}_\perp$ definiert als

$$f_i(x) = \begin{cases} \perp & \text{falls } x \geq i, \\ 0 & \text{falls } x < i. \end{cases}$$

Dann ist $c\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{N}_\perp)$ mit $c(i) = f_i$ eine Kette, wobei $\sqcup c$ die konstante 0-Funktion ist. Deshalb gilt $t(\sqcup c) = 1$, weil $\sqcup c$ total ist. Für jedes $i \in \mathbb{N}$ gilt aber, dass f_i nicht total ist, also ist $t(f_i) = 0$ und somit $\sqcup(t \circ c) = 0$.

Semantik der Fakultät

Sei wieder fact: $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$ definiert als

$$\text{fact}(x) = \begin{cases} \perp & \text{falls } x = \perp, \\ 1 & \text{falls } x < 0, \\ x! & \text{falls } x \geq 0. \end{cases}$$

Eine *rekursive* Implementierung von fact in Haskell könnte folgendermaßen aussehen:

```
fact :: Int -> Int
fact x = if x <= 0 then 1
         else x * fact (x - 1)
```

Wegen dem rekursiven Aufruf hängt die Semantik von `fact` von sich selbst ab. Wie gehen wir damit um?

Semantik der Fakultät

Betrachten wir wieder die Kette $c(i) = \mathsf{fact}_i$, wobei $\mathsf{fact}_i : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ für $i \in \mathbb{N}$ definiert ist als

$$\mathsf{fact}_0(x) = \perp$$

$$\mathsf{fact}_{i+1}(x) = \begin{cases} \perp & \text{falls } x = \perp \text{ oder } x > i, \\ 1 & \text{falls } x < 0, \\ x! & \text{falls } 0 \leq x \leq i. \end{cases}$$

In Haskell könnte man für endlich viele i jeweils fact_i *ohne Rekursion* über `fact_i :: Int -> Int` implementieren mit

```
fact_0 x = undefined
fact_{i+1} x = if x <= 0 then 1
               else x * fact_i (x - 1)
```

Semantik der Fakultät

Wir haben zum Beispiel, dass

```
fact_1 x = if x <= 0 then 1
           else x * fact_0 (x - 1)
fact_2 x = if x <= 0 then 1
           else x * fact_1 (x - 1)
```

`fact_{i+1}` ist also die Funktion, wo die Rekursion i mal aufgefaltet wurde. Deshalb berechnet `fact_{i+1}` die Fakultät „bis i“, also fact_{i+1}. Solch eine Implementierung würde natürlich unendlich viel Code benötigen.
Stattdessen betrachten wir die Funktion `rfact`, die es uns erlaubt, von `fact_i` zu `fact_{i+1}` zu gelangen, also die Rekursion einmal aufzufalten. Diese Funktion erhält im Gegensatz zu `fact` die rekursiv aufzurufende Funktion als Parameter.

Semantik der Fakultät

```
rfact :: (Int -> Int) -> (Int -> Int)
rfact f x = if x <= 0 then 1
            else x * f (x - 1)
```

Wir erhalten für $[\![\mathrm{rfact}]\!] : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to (\mathbb{Z}_\perp \to \mathbb{Z}_\perp)$, dass

$$[\![\mathrm{rfact}]\!](f)(x) = \begin{cases} \perp & \text{falls } x = \perp, \\ 1 & \text{falls } x \leq 0, \\ x \cdot f(x-1) & \text{falls } x > 0. \end{cases}$$

Wir können nun `fact_i` erhalten, indem wir `rfact` i mal auf **`undefined`** anwenden, also

```
fact_0 = undefined
fact_1 = rfact undefined
fact_2 = rfact (rfact undefined)
...
```

Fixpunkte

Die Idee ist nun, dass man immer weiter `rfact` anwendet, also die Rekursion immer weiter auffaltet, und so irgendwann bei `fact` ankommt. Intuitiv ist man mit dem Prozess erst fertig, wenn man eine Funktion f erhält, sodass $[\![\mathrm{rfact}]\!](f) = f$ gilt. Das heißt, dass `f` ein *Fixpunkt* von $[\![\mathrm{rfact}]\!]$ ist. In dem Fall kann nämlich ein weiteres Anwenden von `rfact` nichts „Neues“ mehr bringen.
Die Fakultät ist ein solcher Fixpunkt von $[\![\mathrm{rfact}]\!]$, da sie folgende Gleichung erfüllt:

$$[\![\mathsf{rfact}]\!](\mathsf{fact})(x) = \begin{cases} \bot & \text{falls } x = \bot, \\ 1 & \text{falls } x \leq 0, \\ x \cdot \mathsf{fact}(x-1) & \text{falls } x > 0. \end{cases}$$

Dies kann man nachrechnen, indem man die Definition von fact einsetzt.

Fixpunkte

Funktionen können aber mehrere Fixpunkte haben. Zum Beispiel betrachten wir

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

Wir können auch hier die Rekursion als Parameter übergeben und erhalten:

```
rnonterm :: (Int -> Int) -> (Int -> Int)
rnonterm f x = f (x + 1)
```

Die Forderung für einen Fixpunkt f von $[\![\mathrm{rnonterm}]\!]$ ist also, dass $f(x) = f(x+1)$ für alle $x \in \mathbb{Z}$ gilt. Das heißt, es muss $f(y) = f(z)$ für alle $y, z \in \mathbb{Z}$ gelten. Außerdem gilt $f(\bot) = f(\bot + 1) = f(\bot)$, also wird über $f(\bot)$ nichts gefordert.

Fixpunkte

Wir wollen nun, dass `fact` der „am wenigsten definierte“ Fixpunkt von `rfact` ist. Intuitiv heißt das, dass der Fixpunkt, den wir wählen, nichts definiert, was nicht aus der Definition von `rfact` kommt. Im Fall von `nonterm` ist dies also die überall undefinierte Funktion.

Formal ist ein *Fixpunkt* einer Funktion $f : D \to D$ ein Element $x \in D$ mit $f(x) = x$. Sei $(D, \sqsubseteq_D)$ eine partielle Ordnung. x heißt *kleinster Fixpunkt von f*, wenn er kleiner als alle (anderen) Fixpunkte von f ist, also wenn für jedes $y \in D$ mit $f(y) = y$ gilt, dass $x \sqsubseteq_D y$. Wir bezeichnen den kleinsten Fixpunkt von f bezüglich $\sqsubseteq_D$ (falls dieser existiert) mit μf.

Fixpunktsatz (Teil 1)

Der folgende Satz ist das zentrale Ergebnis über stetige Funktionen. Er besagt, dass jede stetige Funktion einen kleinsten Fixpunkt hat und zeigt sogar, wie man diesen erhält.

Satz 11

Sei $(D, \sqsubseteq_D)$ eine CPO. Jede stetige Funktion $f : [D \to D]$ hat einen kleinsten Fixpunkt. Dieser ist $\mu f = \sqcup c$, wobei $c : \mathbb{N} \to D$ die Kette $c(i) = f^i(\bot_D)$ ist.

Beweis.

Da $\bot_D$ kleinstes Element ist, gilt $\bot_D \sqsubseteq f(\bot_D)$. Per Induktion folgt aus der Monotonie von f, dass $f^i(\bot_D) \sqsubseteq f^{i+1}(\bot_D)$ für alle $i \in \mathbb{N}$. Somit ist c eine Kette. Wir wollen nun zeigen, dass $\sqcup c = \mu f$.

Fixpunktsatz (Teil 2)

Beweis.

Um zu zeigen, dass $\sqcup c$ überhaupt ein Fixpunkt von f ist, müssen wir zeigen, dass $f(\sqcup c) = \sqcup c$. Zunächst gilt, dass $f(\sqcup c) = \sqcup(f \circ c)$, weil f stetig ist. Es gilt $\lambda i.(f \circ c)(i) = \lambda i.f^{i+1}(\bot_D) = \lambda i.c(i+1)$. Dann ist $c' : \mathbb{N} \to D$ mit $c'(i) = c(i+1) = (f \circ c)(i)$ auch eine Kette (nach Folgerung 5). Des Weiteren gilt $\sqcup c = \sqcup c'$ (Beweis: Übung), woraus wir schließen, dass $\sqcup(f \circ c) = \sqcup c$.
Sei d ein (anderer) Fixpunkt von f, also $f(d) = d$. Zu zeigen ist noch, dass $\sqcup c \sqsubseteq d$. Es gilt $\bot_D \sqsubseteq d$. Per Induktion folgt aus der Monotonie von f, dass $f^i(\bot_D) \sqsubseteq f^i(d)$ für alle $i \in \mathbb{N}$. Weil d Fixpunkt ist, gilt auch $f^i(d) = d$ für alle $i \in \mathbb{N}$, also $f^i(\bot_D) \sqsubseteq d$. Damit haben wir, dass $\mathrm{Img}(c) = \{f^i(\bot_D) \mid i \in \mathbb{N}\} \sqsubseteq d$ und somit $\sqcup c \sqsubseteq d$. □

Semantik der Fakultät

Wir wollen nun noch zeigen, dass $\mu[\![\mathrm{rfact}]\!] = \mathsf{fact}$. Dazu zeigen wir zunächst per Induktion, dass $[\![\mathrm{rfact}]\!]^i(\bot) = \mathsf{fact}_i$ für alle $i \in \mathbb{N}$. Es gilt $[\![\mathrm{rfact}]\!]^0(\bot) = \bot = \mathsf{fact}_0$. Sei $i \in \mathbb{N}$. Dann gilt $[\![\mathrm{rfact}]\!](\mathsf{fact}_i) = \mathsf{fact}_{i+1}$ (Beweis: Übung). Aus $[\![\mathrm{rfact}]\!]^i(\bot) = \mathsf{fact}_i$ folgt, dass

$$[\![\mathrm{rfact}]\!]^{i+1}(\bot) = [\![\mathrm{rfact}]\!]([\![\mathrm{rfact}]\!]^i(\bot)) = [\![\mathrm{rfact}]\!](\mathsf{fact}_i) = [\![\mathsf{fact}_{i+1}]\!].$$

Wir erhalten mit dem Fixpunktsatz, dass $\mu[\![\mathrm{rfact}]\!] = \sqcup\lambda i.[\![\mathrm{rfact}]\!]^i(\bot)$. Außerdem wissen wir bereits, dass $\sqcup\lambda i.\,\mathsf{fact}_i = \mathsf{fact}$. Insgesamt erhalten wir also, dass

$$\mu[\![\mathrm{rfact}]\!] = \sqcup\lambda i.[\![\mathrm{rfact}]\!]^i(\bot) = \sqcup\lambda i.\,\mathsf{fact}_i = \mathsf{fact}\,.$$

Man müsste noch zeigen, dass $[\![\mathrm{rfact}]\!]$ stetig ist. Dies erhält man aus der Tatsache, dass Fallunterscheidung, Applikation und die Basisfunktionen wie Vergleiche, Multiplikation, usw. stetig sind.

Fixpunkte

Den kleinsten Fixpunkt von `rfact` kann man *nicht* ausrechnen, da man `rfact` unendlich oft anwenden müsste. Warum ist dies aber kein Problem?
Erstens wollen wir über diesen Prozess eine mathematische Funktion definieren. Dazu müssen wir diese nicht von einem Programm ausrechnen lassen.
Zweitens genügt es für das Ausrechnen einer Stelle der Fakultät, `rfact` nur *endlich* oft anzuwenden, also eine endliche Approximation von $\mu[\![\mathrm{rfact}]\!]$ auszurechnen.

Domains

Um denotationelle Semantik für Haskell-Programme formal zu definieren, müssen wir zunächst die Wertebereiche für Typen festlegen. Diese werden *Domains* genannt. Jede Domain wird eine CPO sein. Dem Basistyp `Int` ordnen wir den Domain $\mathcal{D}(\mathrm{Int}) = \mathbb{Z}_\perp$ zu.
Welche Domains sollte man Data-Deklarationen zuordnen?
Betrachten wir

```
data IntPair = Make Int Int
```

mit folgenden Beispielwerten

```
c :: IntPair
d :: IntPair
c = undefined
d = Make undefined undefined
```

Hier ist `c` komplett undefiniert, wohingegen `d` den Wertkonstruktor `Make` mit zwei undefinierten `Ints` enthält.

Domains von Data-Deklarationen

Dass `c` und `d` sich anders verhalten, kann man gut an den folgenden beiden Funktionen erkennen:

```
f :: IntPair -> Int
f p = 0
g :: IntPair -> Int
g p = case p of Make x y -> 0
```

Wir haben, dass `f c = 0` und `f d = 0`. Ebenso haben wir `g d = 0`. Allerdings terminiert `g c` nicht, weil es sein Argument so weit auswerten muss, bis klar ist, welcher Wertkonstruktor (hier `Make`) angewandt wurde.

Wir benötigen also nicht nur $\perp$-Werte für die Argumente von Wertkonstruktoren, sondern auch noch ein zusätzliches $\perp$ für den Fall, dass kein Wertkonstruktor angewandt wurde.

Weitere CPOs

Um die nächsten CPO-Konstruktionen etwas zu vereinfachen, formulieren wir folgendes Lemma:

Lemma 12

Sei $(D, \sqsubseteq_D)$ eine CPO und $c : \mathbb{N} \to D$ eine Kette.

1. *Dann ist auch $c_k : \mathbb{N} \to D$ mit $c_k(n) = c(n+k)$ für jedes $k \in \mathbb{N}$ eine Kette und es gilt $\sqcup c_k = \sqcup c$.*
2. *Sei $(E, \sqsubseteq_E)$ eine CPO mit $E \subseteq D$, wobei $\sqsubseteq_E$ die Einschränkung von $\sqsubseteq_D$ auf E ist, und sei $\mathrm{Img}(c) \subseteq E$. Dann ist auch $c_E : \mathbb{N} \to E$ mit $c_E(n) = c(n)$ eine Kette und es gilt $\sqcup c_E = \sqcup c$.*

Beweis.

Wir haben Punkt 1 bereits für $k = 1$ in der Übung gezeigt. Für $k > 1$ funktioniert die Argumentation analog. Punkt 2 ist klar. □

Lift einer CPO (Teil 1)

Der *Lift einer CPO* $(D, \sqsubseteq_D)$ ist $(D_\perp, \sqsubseteq_{D_\perp})$ mit $D_\perp = D \uplus \{\perp_D\}$ und $d \sqsubseteq_{D_\perp} d'$ genau dann, wenn $d = \perp_D$ oder $d, d' \in D$ mit $d \sqsubseteq_D d'$.

Lemma 13

$(D_\perp, \sqsubseteq_{D_\perp})$ *ist eine CPO.*

Beweis.

Reflexivität: Sei $d \in D_\perp$. Wenn $d = \perp_D$, dann gilt $d \sqsubseteq d$. Wenn $d \in D$, dann gilt $d \sqsubseteq_D d$, also auch $d \sqsubseteq d$. Antisymmetrie: Seien $d, d' \in D_\perp$ mit $d \sqsubseteq d'$ und $d' \sqsubseteq d$. Wenn $d = \perp_D$, dann gilt $d' \sqsubseteq \perp_D$ und somit muss $d' = \perp_D$ sein. Der Fall, dass $d' = \perp_D$, ist analog. Wenn $d, d' \in D$, dann gilt $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d$, also auch $d = d'$. Transitivität: Seien $d, d', d'' \in D_\perp$ mit $d \sqsubseteq d'$ und $d' \sqsubseteq d''$. Wenn $d = \perp_D$, dann gilt auch $d \sqsubseteq d''$. Wenn $d \in D$, dann gilt $d \sqsubseteq_D d'$. Daraus folgt $d' \in D$, also auch $d' \sqsubseteq_D d''$. Also gilt $d' \sqsubseteq_D d''$ und somit $d \sqsubseteq d''$.

Lift einer CPO (Teil 2)

Beweis.

Nach Definition von $\sqsubseteq_{D_\perp}$ ist klar, dass $\perp_D$ das kleinste Element ist. Sei $c\colon \mathbb{N} \to D_\perp$ eine Kette. Wir haben zwei Fälle zu unterscheiden: Wenn $c(n) = \perp_D$ für alle $n \in \mathbb{N}$, dann ist $c'\colon \mathbb{N} \to \{\perp_D\}$ mit $c'(d) = c(d)$ eine Kette und es gilt $\sqcup c = \sqcup c'$ nach Punkt 2 von Lemma 12. Ansonsten gibt es ein $k \in \mathbb{N}$ mit $c(k) \in D$. Nach Definition von $\sqsubseteq_{D_\perp}$ muss dann für alle $k' > k$ gelten, dass $c(k') \in D$. Also ist $c'\colon \mathbb{N} \to D$ mit $c'(n) = c(n + k)$ eine Kette und es gilt $\sqcup c = \sqcup c'$ nach Punkten 2 und 1 von Lemma 12. □

Domains von Data-Deklarationen

Wir betrachten zunächst nicht rekursive Data-Deklarationen mit nur einem Wertkonstruktor: `data TCon = VCon`. Solch ein Wertkonstruktor `VCon` ist allgemein von der Form

```
VCon = N T_1 ... T_n
```

wobei `N` ein Bezeichner ist und `T_1` bis `T_n` für $n \geq 0$ Typen sind. Werte, die mit `N` erzeugt wurden, sollen ein Element folgender Menge sein:

$$\mathcal{D}(\mathrm{VCon}) := \{\underline{\mathrm{N}}\} \times \mathcal{D}(\mathrm{T_1}) \times \cdots \times \mathcal{D}(\mathrm{T_n})$$

Die Menge $\{\underline{\mathrm{N}}\}$ enthält nur einen Wert, der uns sagt, welcher Wertkonstruktor angewandt wurde. Da wir ein zusätzliches $\bot$ benötigen für den Fall, dass kein Wertkonstruktor angewandt wurde, wählen wir für `TCon` also

$$\mathcal{D}(\mathrm{TCon}) := \mathcal{D}(\mathrm{VCon})_\bot = (\{\underline{\mathrm{N}}\} \times \mathcal{D}(\mathrm{T_1}) \times \cdots \times \mathcal{D}(\mathrm{T_n}))_\bot$$

Domains von Data-Deklarationen

Betrachten wir wieder

```
data IntPair = Make Int Int
```

Hier haben wir also

$$\mathcal{D}(\text{IntPair}) = \mathcal{D}(\text{Make Int Int})_\perp = (\{\underline{\text{Make}}\} \times \mathbb{Z}_\perp \times \mathbb{Z}_\perp)_\perp$$

Für die vorherigen Werte

```
c = undefined
d = Make undefined undefined
```

erhalten wir dann $[\![\text{c}]\!] = \perp$ und $[\![\text{d}]\!] = (\underline{\text{Make}}, \perp, \perp)$. Für

```
e = Make 0 1
```

erhalten wir $[\![\text{e}]\!] = (\underline{\text{Make}}, 0, 1)$.

Domains für mehrere Wertkonstruktoren

Als Nächstes betrachten wir Data-Deklarationen mit mehr als einem Wertkonstruktor. Die Einfachste von diesen ist `Bool` mit

```
data Bool = True | False
```

Wir haben $\mathcal{D}(\mathrm{True}) = \{\underline{\mathrm{True}}\}$ und $\mathcal{D}(\mathrm{False}) = \{\underline{\mathrm{False}}\}$ für die beiden Wertkonstruktoren. Um beide Werte zusammenzufügen, liften wir zunächst die Domains und erhalten $\mathcal{D}(\mathrm{True})_\perp = \{\underline{\mathrm{True}}, \perp\}$ und $\mathcal{D}(\mathrm{False})_\perp = \{\underline{\mathrm{False}}, \perp\}$. Diese werden dann über die verschmolzene Summe zusammengefügt, wo die beiden $\perp$ von $\mathcal{D}(\mathrm{True})_\perp$ und $\mathcal{D}(\mathrm{False})_\perp$ identifiziert werden. Wir werden außerdem aus technischen Gründen Elemente aus $\mathcal{D}(\mathrm{True})$ und $\mathcal{D}(\mathrm{False})$ mit „Tags“ versehen, die sicherstellen, dass nur disjunkte Mengen vereinigt werden.

Summe

Die *Summe* von $n \geq 0$ Mengen $M_1, \ldots, M_n$ ist

$$M_1 + \cdots + M_n = (M_1 \times \{1\}) \cup \cdots \cup (M_n \times \{n\}).$$

Im Fall, dass $n = 0$, nehmen wir $\emptyset$. Die „Tags“ $1, \ldots, n$ stellen sicher, dass man für jedes Element weiß, aus welcher Menge es gekommen ist, da die Mengen $M_1, \ldots, M_n$ nicht disjunkt sein müssen. Im Gegensatz zu Summen kann man bei der Vereinigung zweier Mengen diese Information verlieren, zum Beispiel bei $\mathbb{N} \cup \mathbb{Z} = \mathbb{Z}$. Der Name Summe kommt daher, dass $|A + B| = |A| + |B|$ für alle Mengen A und B gilt. Wenn klar ist, aus welcher Quellmenge ein Element $(m, i) \in M_i \times \{i\}$ kommt, schreiben wir auch einfach nur m.

Verschmolzene Summe

Seien $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ CPOs. Die *Verschmolzene Summe* $(D_1 \oplus \cdots \oplus D_n, \sqsubseteq_{D_1 \oplus \cdots \oplus D_n})$ ist

$$D_1 \oplus \cdots \oplus D_n = ((D_1 \setminus \{\bot_{D_1}\}) + \cdots + (D_n \setminus \{\bot_{D_n}\})) \cup \{\bot_{D_1 \oplus \cdots \oplus D_n}\},$$

wobei $d \sqsubseteq_{D_1 \oplus \cdots \oplus D_n} d'$ genau dann, wenn $d = \bot_{D_1 \oplus \cdots \oplus D_n}$, oder $d = (e, i) \in (D_i \setminus \{\bot_{D_i}\}) \times \{i\}$ und $d' = (e', i) \in (D_i \setminus \{\bot_{D_i}\}) \times \{i\}$ für $1 \leq i \leq n$ mit $e \sqsubseteq_{D_i} e'$.
Wir entfernen also alle kleinsten Elemente aus $D_1, \ldots, D_n$ und fügen ein neues kleinstes Element hinzu.

Lemma 14

$(D_1 \oplus \cdots \oplus D_n, \sqsubseteq_{D_1 \oplus \cdots \oplus D_n})$ *ist eine CPO.*

Beweis.

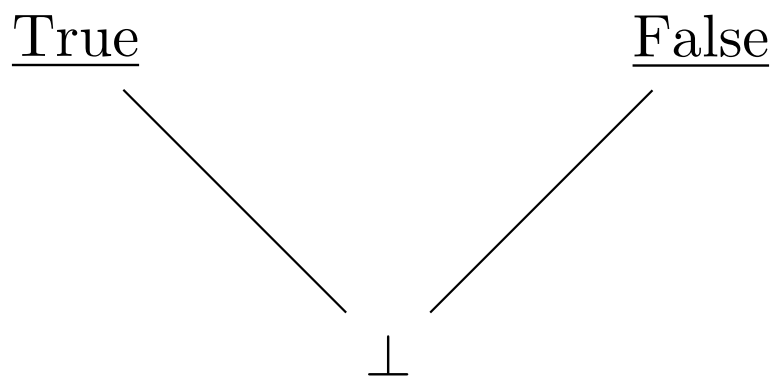
Übung. □

Domains für mehrere Wertkonstruktoren

Im Fall von `Bool` erhalten wir also

$$\begin{aligned}\mathcal{D}(\mathrm{Bool}) &= \mathcal{D}(\mathrm{True})_\perp \oplus \mathcal{D}(\mathrm{False})_\perp \\ &= \{\underline{\mathrm{True}}, \perp\} \oplus \{\underline{\mathrm{False}}, \perp\} \\ &= (\{\underline{\mathrm{True}}\} + \{\underline{\mathrm{False}}\}) \cup \{\perp\} \\ &= \{(\underline{\mathrm{True}}, 1), (\underline{\mathrm{False}}, 2), \perp\}\end{aligned}$$

Da bei $(\underline{\mathrm{True}}, 1)$ und $(\underline{\mathrm{False}}, 2)$ klar ist, aus welcher Menge sie kamen, können wir stattdessen einfach $\underline{\mathrm{True}}$ und $\underline{\mathrm{False}}$ schreiben. Grafisch ergibt sich folgende Ordnung:



Domains für mehrere Wertkonstruktoren

Allgemeiner sei

```
data TCon = VCon_1 | ... | VCon_n
```

wobei jedes `VCon_i` für $1 \leq i \leq n$ mit $n \geq 0$ einen Wertkonstruktor deklariert. Auch hier erlauben wir zunächst keine Rekursion. Wir ordnen `TCon` folgenden Domain zu:

$$\mathcal{D}(\mathrm{TCon}) = \mathcal{D}(\mathrm{VCon_1})_\perp \oplus \cdots \oplus \mathcal{D}(\mathrm{VCon_n})_\perp$$

Da wir im Allgemeinen nur Fälle betrachten werden, wo die Wertkonstruktornamen in den `VCon_i` verschieden sind, werden wir die Tags in der Summe – wie wir es bei $\underline{\mathrm{True}}$ und $\underline{\mathrm{False}}$ getan haben – ignorieren können.

Domains für rekursive Data-Deklarationen

Wir müssen nun noch rekursiven Data-Deklarationen und Data-Deklarationen mit Typparametern Domains zuordnen. Diese Konstruktionen sind mathematisch sehr aufwändig und können hier leider nur angedeutet werden.

Betrachten wir wieder

```
data Nat = Zero | Succ Nat
```

Dies führt zu der Gleichung

$$\mathcal{D}(\mathrm{Nat}) = \{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \mathcal{D}(\mathrm{Nat}))_\perp.$$

Ähnlich wie bei rekursiv definierten Funktionen lösen wir diese Gleichung, indem wir nach einem Fixpunkt suchen. Wir fangen mit $\{\perp\}$ für $\mathcal{D}(\mathrm{Nat})$ an und wenden dann wiederholt die folgende Operation an:

$$\lambda D.\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times D)_\perp$$

Domains für rekursive Data-Deklarationen

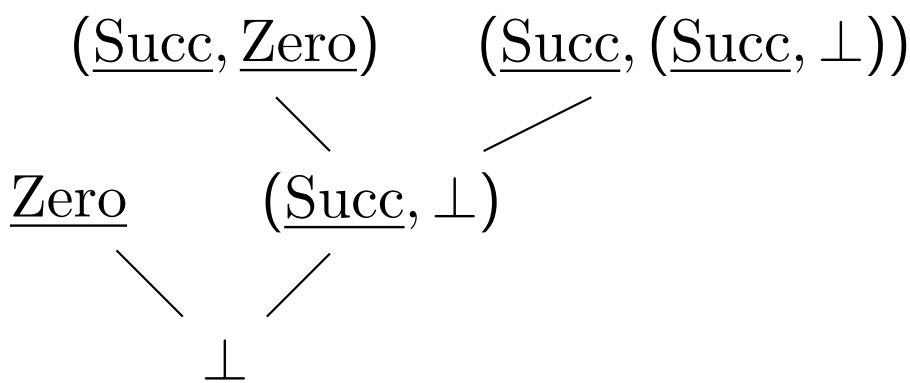
Wie bei $\mathcal{D}(\mathrm{Bool})$ lassen wir wieder die Tags weg. Im ersten Schritt erhalten wir also

$$\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \{\perp\})_\perp = \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp)\}$$

Setzen wir dies wieder ein, erhalten wir

$$\begin{aligned}&\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp)\})_\perp \\ &= \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp), (\underline{\mathrm{Succ}}, \underline{\mathrm{Zero}}), (\underline{\mathrm{Succ}}, (\underline{\mathrm{Succ}}, \perp))\}\end{aligned}$$

Grafisch ergibt sich folgende Ordnung:



Domains für rekursive Data-Deklarationen

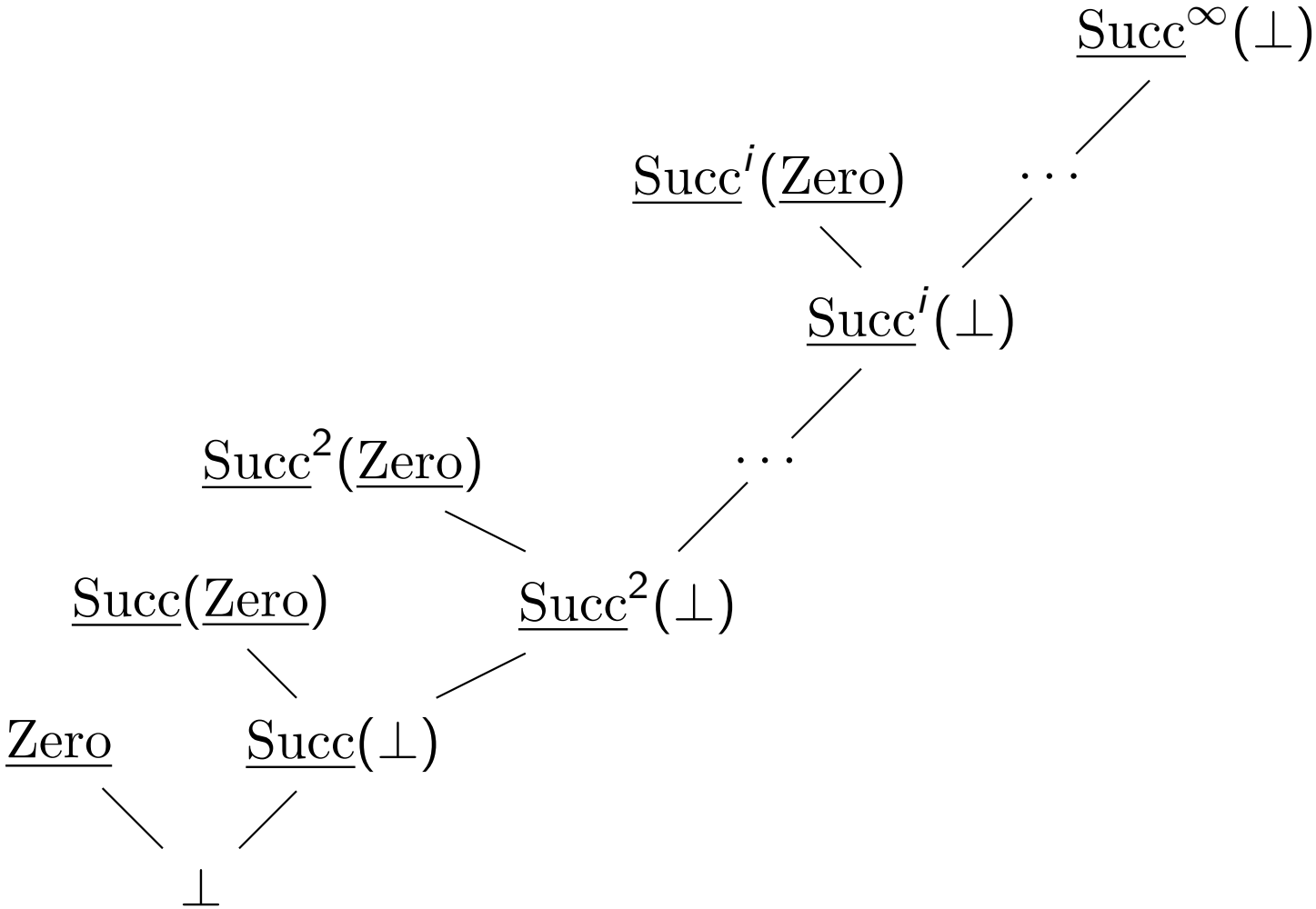
Wir schreiben nun $\underline{\text{Succ}}^n(x)$ statt $\underbrace{(\underline{\text{Succ}}, \cdots (\underline{\text{Succ}},}_{n \text{ mal } (\underline{\text{Succ}},} \underline{\text{x}} \underbrace{) \cdots)}_{n \text{ mal })}$ für $n \in \mathbb{N}$. Jedes $\underline{\text{Succ}}^i(\underline{\text{Zero}})$ für $i \in \mathbb{N}$ kann man einfach über ein Haskell-Programm erzeugen, das i mal `Succ` auf `Zero` anwendet. Ebenso kann man einfach $\underline{\text{Succ}}^i(\bot)$ erzeugen. Wir können allerdings noch einen weiteren Wert erzeugen. Betrachten wir:

```
inf :: Nat
inf = Succ inf
```

Die Semantik von `inf` kann nicht gleich $\underline{\text{Succ}}^i(\bot)$ für ein $i \in \mathbb{N}$ sein, weil man es $i + 1$ mal auf `Succ` matchen kann. Wir benötigen also noch einen weiteren Wert $\underline{\text{Succ}}^\infty(\bot)$, der für unendlich viele Anwendungen von $\underline{\text{Succ}}$ steht.

Domains für rekursive Data-Deklarationen

Grafisch ergibt sich also folgender Domain:



Domains für Data-Deklarationen mit Parametern

Betrachten wir als Beispiel wieder

```
data List a = Nil | Cons a (List a)
```

Für einen konkreten Typ `t` wählen wir dann $\mathcal{D}(\mathrm{List}(\mathrm{t}))$ als den kleinsten Fixpunkt von der Operation

$$\lambda D.\{\underline{\mathrm{Nil}}\}_\perp \oplus (\{\underline{\mathrm{Cons}}\} \times \mathcal{D}(\mathrm{t}) \times D)_\perp.$$

Ein Wert, der den Typ `List a` hat, wobei `a` eine Typvariable ist, hat auch den Typ `List t` für *alle* Typen `t`. Solch ein Wert ist zum Beispiel `Nil`. Wir wählen daher

$$\mathcal{D}(\mathrm{List}(\mathrm{a})) = \bigcap \{\mathcal{D}(\mathrm{List}(\mathrm{t})) \mid t \text{ ist ein Typ}\}.$$

Welche Typen es gibt, hängt allerdings davon ab, welche Data-Deklarationen in einem Programm vorkommen.

Domain eines Programms

Zu $n \in \mathbb{N}$ sei Con_n die Menge aller n-stelligen Wertkonstruktoren, die in einem Programm vorkommen. Dann sei

$$\mathrm{Functions} = [\mathrm{Dom} \to \mathrm{Dom}]_\perp \text{ und}$$
$$\mathrm{Constructors}_n = (\mathrm{Con}_n \times \mathrm{Dom}^n)_\perp \text{ für } n \in \mathbb{N},$$

wobei wir für *den Domain* Dom *des Programms* die kleinste Menge nehmen, die folgende Gleichung erfüllt:

$$\mathrm{Dom} = \mathbb{Z}_\perp \oplus \mathrm{Functions} \oplus \mathrm{Constructors}_0 \oplus \mathrm{Constructors}_1 \oplus \cdots$$

Diese Menge enthält auch nicht typkorrekte Werte wie $(\underline{\mathrm{Succ}}, \underline{\mathrm{True}})$, die wegen Typüberprüfung zur Laufzeit nicht vorkommen können.
Einige Funktionen, die wir für die Semantik brauchen, werden der Einfachheit halber den Domain des Programms als Definitions- bzw. Zielbereich haben.

Syntax von Haskell

Um formal die Semantik eines Haskell-Programms festzulegen, müssen wir zunächst definieren, welche Syntax wir erlauben. Wir werden hierbei einige Einschränkungen vornehmen, die allerdings keine wirklichen Einschränkungen sind, da man alle anderen Haskell-Programme in unsere erlaubte Syntax übersetzen kann.

Der Einfachheit nehmen wir an, dass ein Haskell-Programm aus Deklarationen folgender Form besteht:

$$\underline{\text{decl}} \to \underline{\text{var}} = \underline{\text{exp}}$$

Hier steht $\underline{\text{var}}$ für alle Variablennamen. Die Menge aller Variablen bezeichnen wir mit Var.

Beispiele für solche Deklarationen sind `x = 5` und
`f = \x -> \y -> x + y`.

Syntax von Haskell

Die erlaubten Ausdrücke, die wir mit Exp bezeichnen, sind durch folgende Grammatik gegeben:

$$
\begin{aligned}
\underline{\text{exp}} \rightarrow\ & \underline{\text{var}} \\
\mid\ & \underline{\text{constr}} \\
\mid\ & \underline{\text{integer}} \\
\mid\ & (\underline{\text{exp}}\ \underline{\text{exp}}) \\
\mid\ & \textbf{let}\ \underline{\text{var}} = \underline{\text{exp}}\ \textbf{in}\ \underline{\text{exp}} \\
\mid\ & \backslash\underline{\text{var}} \texttt{->} \underline{\text{exp}} \\
\mid\ & \textbf{case}\ \underline{\text{exp}}\ \textbf{of}\ \{\underline{\text{pat}} \texttt{->} \underline{\text{exp}}\ ;\ \ldots\ ;\ \underline{\text{pat}} \texttt{->} \underline{\text{exp}}\}
\end{aligned}
$$

Hierbei erkennt $\underline{\text{constr}}$ alle Wertkonstruktornamen, also die Elemente aus allen Con_n für $n \in \mathbb{N}$, und $\underline{\text{integer}}$ alle ganzen Zahlen. Die Grammatik für $\underline{\text{pat}}$ ist

$$\underline{\text{pat}} \rightarrow \underline{\text{constr}}\ \underline{\text{var}}\ \ldots\ \underline{\text{var}}$$

Syntax von Haskell

($\underline{\text{exp}}$ $\underline{\text{exp}}$) ist die Applikation, also das Anwenden eines Parameters auf eine Funktion. Wir haben meistens die Klammern weggelassen, da vereinbart wird, dass man statt $(\cdots(e_1\ e_2)\cdots e_n)$ auch $e_1 \cdots e_n$ schreiben darf.
Operatoren behandeln wir wie folgt: Statt die Infixschreibweise wie `2 + 5` zu verwenden, muss man die Präfixschreibweise `(((+) 2) 5)` verwenden. Hierbei ist `(+)` eine bereits bekannte Variable mit vordefinierter Semantik.
Statt Case-Ausdrücke einzurücken, müssen wir {, } und ; verwenden. Zum Beispiel steht

```
case e of { [] -> 0 ; x : xs -> 1 }
```

für

```
case e of
  [] -> 0
  x : xs -> 1
```

Umgebung

Wir nennen eine partielle Funktion $\eta\colon \text{Var} \to \text{Dom}$ eine *Umgebung*. Wir gehen davon aus, dass gewisse Operationen wie + in jeder Umgebung eingetragen sind, also zum Beispiel soll gelten, dass $\eta((+)) = f_+$, wobei $f_+\colon \text{Dom} \to \text{Dom} \to \text{Dom}$ mit

$$f_+(x)(y) = \begin{cases} x + y & \text{falls } x, y \in \mathbb{Z}, \\ \bot & \text{sonst.} \end{cases}$$

Die *Konkatenation* zweier Umgebungen η_1 und η_2 ist $(\eta_1 \oslash \eta_2)\colon \text{Var} \to \text{Dom}$ mit

$$(\eta_1 \oslash \eta_2)(x) = \begin{cases} \eta_2(x) & \text{falls } \eta_2(x) \text{ definiert ist,} \\ \eta_1(x) & \text{sonst.} \end{cases}$$

Das heißt, dass η_2 Einträge in η_1 überschreibt.

Freie Variablen

Wir können einem Ausdruck nur eine Semantik zuordnen, wenn wir bereits eine Semantik für alle in ihm *frei vorkommenden Variablen* haben. Wir definieren dazu die Funktion free: $\mathsf{Exp} \to 2^{\mathsf{Var}}$ mit

$$
\begin{aligned}
\mathsf{free}(x) &= \{x\} \text{ für } x \in \mathsf{Var},\\
\mathsf{free}(\underline{\mathrm{C}}) &= \emptyset \text{ für } \underline{\mathrm{C}} \in \mathsf{Con}_n, n \in \mathbb{N},\\
\mathsf{free}(i) &= \emptyset \text{ für } i \in \mathbb{Z},\\
\mathsf{free}((e_1\ e_2)) &= \mathsf{free}(e_1) \cup \mathsf{free}(e_2),\\
\mathsf{free}(\textbf{let } x = e_1 \textbf{ in } e_2) &= (\mathsf{free}(e_1) \cup \mathsf{free}(e_2)) \setminus \{x\},\\
\mathsf{free}(\backslash x \mathrel{->} e) &= \mathsf{free}(e) \setminus \{x\}.
\end{aligned}
$$

Für $e = \textbf{case } e' \textbf{ of } \{p_1 \mathrel{->} e_1 \,;\, \cdots \,;\, p_n \mathrel{->} e_n\}$ nehmen wir

$$\mathsf{free}(e) = \mathsf{free}(e') \cup \bigcup \{\mathsf{free}(e_i) \setminus \mathsf{free}(p_i)) \mid 1 \leq i \leq n\}.$$

Die freien Variablen eines Patterns $p = \underline{\mathrm{C}}\ x_1 \ldots x_n$ sind $\mathsf{free}(p) = \{x_1, \ldots, x_n\}$.

Semantik

Für eine Umgebung η mit Definitionsbereich $\{x_1, \ldots, x_n\}$, wobei $\eta(x_i) = d_i$ für $1 \leq i \leq n$, schreiben wir einfach $[x_1/d_1, \ldots, x_n/d_n]$.

Zu einem Ausdruck $e \in \text{Exp}$ und einer Umgebung $\eta\colon \text{Var} \to \text{Dom}$ können wir nun *die Semantik* $[\![\mathrm{e}]\!]_\eta \in \text{Dom}$ definieren, falls $\eta(x)$ für alle $x \in \text{free}(e)$ definiert ist. Dafür machen wir eine Fallunterscheidung über die möglichen Ausdrücke.

Für Variablen $x \in \text{Var}$ sei $[\![\mathrm{x}]\!]_\eta = \eta(x)$. Das heißt, dass Variablen einfach in der Umgebung nachgeschlagen werden. Hier ist natürlich wichtig, dass η auf x definiert ist.

Für Integerwerte $i \in \mathbb{Z}$ sei $[\![\mathrm{i}]\!]_\eta = i$.

Für die Applikation von zwei Ausdrücken, also das Übergeben eines Parameters an eine Funktion, sei $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = [\![\mathrm{e}_1]\!]_\eta([\![\mathrm{e}_2]\!]_\eta)$, falls $[\![\mathrm{e}_1]\!]_\eta\colon \text{Dom} \to \text{Dom}$. Ansonsten ist $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = \bot$. Letzteres bedeutet, dass $[\![\mathrm{e}_1]\!]_\eta$ keine Funktion ist, zum Beispiel bei $[\![(1\ 2)]\!]_\eta$.

Semantik

Für einen Konstruktor $\underline{\mathrm{C}} \in \mathrm{Con}_0$ sei $[\![\underline{\mathrm{C}}]\!]_\eta = \underline{\mathrm{C}}$. Zum Beispiel gilt $[\![\underline{\mathrm{True}}]\!]_\eta = \underline{\mathrm{True}}$.
Für einen Konstruktor $\underline{\mathrm{C}} \in \mathrm{Con}_n$ für $n > 0$ sei $[\![\underline{\mathrm{C}}]\!] = f$ die Funktion $f\colon \underbrace{\mathrm{Dom} \to \cdots \to \mathrm{Dom}}_{n \text{ mal}} \to \mathrm{Dom}$ mit $f(d_1) \cdots (d_n) = (\underline{\mathrm{C}}, d_1, \ldots, d_n)$. Das heißt, dass $[\![\underline{\mathrm{C}}]\!]$ eine n-stellige Funktion ist, die ein Element aus $\mathrm{Constructors}_n$ liefert. Zum Beispiel gilt $[\![\underline{\mathrm{Succ}}]\!]_\eta\colon \mathrm{Dom} \to \mathrm{Dom}$ mit $[\![\underline{\mathrm{Succ}}]\!]_\eta(x) = (\underline{\mathrm{Succ}}, x)$.

Für Lambda-Ausdrücke sei $[\![\backslash \mathrm{x} -> \mathrm{e}]\!]_\eta = f$, wobei $f\colon \mathrm{Dom} \to \mathrm{Dom}$ die Funktion ist mit $f(d) = [\![\mathrm{e}]\!]_{\eta \oplus [x/d]}$. Wir müssen also den Wert für den Parameter x in die Umgebung η eintragen. Zum Beispiel gilt $[\![\backslash \mathrm{x} -> (\underline{\mathrm{Succ}}\ \mathrm{x})]\!]_\eta = f$, wobei $f(d) = [\![(\underline{\mathrm{Succ}}\ \mathrm{x})]\!]_{\eta \oplus [x/d]} = (\underline{\mathrm{Succ}}, d)$.

Semantik

In unserer vereinfachten Sprache sind Let-Ausdrücke der einzige Weg, rekursive Funktionen zu definieren. Bei **let** $x = e_1$ **in** e_2 wird x sowohl in e_1, als auch in e_2 gebunden. Zum Beispiel sei

$$e := \textbf{let } \mathrm{fact} = \backslash x \text{ -> } \textbf{case } x \leq 0 \textbf{ of}$$
$$\{\underline{\mathrm{True}} \text{ -> } 1 \,;\, \underline{\mathrm{False}} \text{ -> } x \cdot \mathrm{fact}\ (x-1)\} \textbf{ in } \mathrm{fact}\ 2$$

Um `fact` eine Semantik im Ausdruck `fact 2` zu geben, bilden wir den kleinsten Fixpunkt der Funktion $f : \mathrm{Dom} \to \mathrm{Dom}$ mit

$$f(d) = [\![\backslash x \text{ -> } \textbf{case } x \leq 0 \textbf{ of}$$
$$\{\underline{\mathrm{True}} \text{ -> } 1 \,;\, \underline{\mathrm{False}} \text{ -> } x \cdot \mathrm{fact}\ (x-1)\}]\!]_{\eta \oplus [\mathrm{fact}/d]}.$$

Dann definieren wir $[\![\mathrm{e}]\!]_\eta = [\![\mathrm{fact}\ 2]\!]_{\eta \oplus [\mathrm{fact}/\mu f]}$.

Allgemein sei $[\![\textbf{let } \mathrm{x} = \mathrm{e}_1 \textbf{ in } \mathrm{e}_2]\!]_\eta = [\![\mathrm{e}_2]\!]_{\eta \oplus [x/\mu f]}$, wobei $f : \mathrm{Dom} \to \mathrm{Dom}$ mit $f(d) = [\![\mathrm{e}_1]\!]_{\eta \oplus [x/d]}$.

Semantik

Sei $e = \textbf{case}\ e'\ \textbf{of}\ \{p_1 \texttt{->} e_1\ ;\ \cdots\ ;\ p_n \texttt{->} e_n\}$. Hier gehen wir wie folgt vor: Haskell versucht zunächst, e' so weit auszuwerten, bis klar ist, welcher Konstruktor angewandt wurde. In unserer Semantik bedeutet das, dass $[\![\mathrm{e}']\!]_\eta = (\underline{\mathrm{C}}, d_1, \ldots, d_m)$ sein muss, wobei $\underline{\mathrm{C}} \in \mathrm{Con}_m$ für ein $m \in \mathbb{N}$ und $d_1, \ldots, d_m \in \mathrm{Dom}$. Ist dies nicht der Fall, dann ist $[\![\mathrm{e}]\!]_\eta = \bot$. Man beachte, dass es nicht vorkommen kann, dass $[\![\mathrm{e}']\!]_\eta = (\underline{\mathrm{C}}, d_1, \ldots, d_m)$, aber $\underline{\mathrm{C}} \in \mathrm{Con}_n$ mit $n \neq m$. Das liegt daran, dass $[\![\underline{\mathrm{C}}]\!]$ für $\underline{\mathrm{C}} \in \mathrm{Con}_m$ immer genau m Argumente erwartet.

Der nächste Schritt ist, den ersten Pattern (von links nach rechts) rauszusuchen, der $\underline{\mathrm{C}}$ verwendet. Hierbei ist es wichtig, dass der Pattern genau m Variablen enthält. Gibt es keinen solchen Pattern, so ist auch $[\![\mathrm{e}]\!]_\eta = \bot$. Ansonsten sei $p_i = \underline{\mathrm{C}}\ x_1\ \ldots\ x_m$ (wobei $1 \leq i \leq n$) dieser erste Pattern. Dann müssen wir d_j für x_j in die Umgebung einsetzen (für $1 \leq j \leq m$) und darunter e_i auswerten.

Semantik

Wir schreiben $C(p_i)$ für den Konstruktor, der in p_i steht, also $C(p_i) = \underline{\mathrm{C_i}}$, falls $p_i = (\underline{\mathrm{C_i}}, x_1, \ldots, x_m)$ für $x_1, \ldots, x_m \in \mathrm{Var}$ und $\underline{\mathrm{C_i}} \in \mathrm{Con}_m$. Dann definieren wir die Semantik von e als

$$[\![\mathrm{e}]\!]_\eta = \begin{cases} [\![\mathrm{e_i}]\!]_{\eta \oplus [x_1/d_1,\ldots,x_m/d_m]} & \text{falls } [\![\mathrm{e'}]\!]_\eta = (\underline{\mathrm{C_i}}, d_1, \ldots, d_m), \\ & \quad p_i = \underline{\mathrm{C_i}}\ x_1\ \ldots\ x_m \text{ und} \\ & \quad C(p_1), \ldots, C(p_{i-1}) \neq \underline{\mathrm{C_i}}, \\ \bot & \text{sonst.} \end{cases}$$

Beispiel: Sei $e = \textbf{case}\ e'\ \textbf{of}\ \{\underline{\mathrm{Zero}} -> \underline{\mathrm{Zero}}\,;\, \underline{\mathrm{Succ}}\ x -> x\}$. Im Fall, dass $[\![\mathrm{e'}]\!]_\eta = \bot$, ist auch $[\![\mathrm{e}]\!]_\eta = \bot$. Wenn $[\![\mathrm{e'}]\!]_\eta = \underline{\mathrm{True}}$, dann ist auch $[\![\mathrm{e}]\!]_\eta = \bot$, weil $\underline{\mathrm{True}}$ nicht auf $\underline{\mathrm{Zero}}$ oder $\underline{\mathrm{Succ}}$ passt. Wenn $[\![\mathrm{e'}]\!]_\eta = (\underline{\mathrm{Succ}}, \underline{\mathrm{Zero}})$, dann ist $[\![\mathrm{e}]\!]_\eta = [\![\mathrm{x}]\!]_{\eta \oplus [x/\underline{\mathrm{Zero}}]} = \underline{\mathrm{Zero}}$.

Semantik

Wir müssten eigentlich noch zeigen, dass alle Funktionen, die wir in der Definition der Semantik benutzt haben, auch stetig sind. Da dies allerdings sehr aufwändig ist, müssen wir aus zeitlichen Gründen leider darauf verzichten.

Für ein gesamtes Programm nehmen wir an, dass es die Form $\underline{\text{decl}} \ \ldots \ \underline{\text{decl}} \ \underline{\text{expr}}$ hat, also $x_1 = e_1 \ \ldots \ x_n = e_n \ e$. Dies können wir schreiben als **let** $x_1 = e_1$ **in** $x_2 = e_2$ **in** $\ldots$ **let** $x_n = e_n$ **in** e. Zum Beispiel können wir statt

```
f = \x -> \y -> x + y
g = \x -> f x 2
f 3
```

dann schreiben

```
let f = \x -> \y -> x + y
in let g = \x -> f x 2
in f 3
```

Typen

Abschließend wollen wir uns noch mit Typen beschäftigen. In unserer Semantik gibt es momentan viele Stellen, die zu „Laufzeitfehlern“ führen, welche man aber bei der Typüberprüfung schon ausschließen kann.

Zunächst wollen wir sicherstellen, dass bei $[\![\mathrm{e}]\!]_\eta$ jedes $x \in \mathrm{free}(e)$ auch einen Eintrag in η hat. Dazu müssen wir verlangen, dass jede Variable auch deklariert wurde.

Bei fundamentalen Funktionen wie $(+)$ gilt $[\![\mathrm{x} + \mathrm{y}]\!]_\eta = \bot$, wenn $[\![\mathrm{x}]\!]_\eta \notin \mathbb{Z}_\bot$ oder $[\![\mathrm{y}]\!]_\eta \notin \mathbb{Z}_\bot$, also zum Beispiel bei $[\![\mathrm{x}]\!]_\eta = \underline{\mathrm{True}}$. Dies können wir vermeiden, indem wir sicherstellen, dass nur Werte vom Typ Int an $(+)$ übergeben werden.

Typen

Bei der Applikation gilt $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = \bot$, wenn $[\![\mathrm{e}_1]\!]_\eta \notin [\mathrm{Dom} \to \mathrm{Dom}]$, also wenn $[\![\mathrm{e}_1]\!]_\eta$ keine Funktion ist. Wir müssen deshalb sicherstellen, dass e_1 einen Funktionstyp hat und dass e_2 den passenden Argumenttyp hat. Zum Beispiel sollte bei `f x y = x + y` die Funktion `f` den Typ `Int -> Int -> Int` bekommen. Ein Programm wie `f True 0` sollte also abgelehnt werden.

Bei **case** erhalten wir $\bot$, wenn die Patterns nicht die richtige Anzahl an Variablen haben. Außerdem wollen wir „unsinnige“ Patterns ausschließen. Beispiel:

```
f x = case x of { True -> 0 ; Nil -> False }
```

Hier kann `x` nicht sowohl den Typ `Bool`, als auch den Typ `List t` für einen Typ `t` haben. Zweitens kann `f` nicht mehr zur Laufzeit entscheiden, ob es einen Wert vom Typ `Int` oder `Bool` liefert.

Typen

Sprachen, die beim Kompilieren eines Programms Typüberprüfung machen, nennt man *statisch typisierte* Sprachen. Die Idee dabei ist, dass viele unsinnige Programme direkt abgelehnt werden und man nicht mühsam nach den dadurch verursachten Fehlern zur Laufzeit suchen muss.
Wegen der Unentscheidbarkeit des Halteproblems können wir nicht genau die Programme ablehnen, die zu Laufzeitfehlern führen.
Unser Typsystem wird sicherstellen, dass alle Programme mit Laufzeitfehlern abgelehnt werden, aber auch andere. Dies ist nötig, damit Wohlgetyptheit entscheidbar ist. Das Programm

```
let e = case True of
    { True -> 0 ; False -> False }
in e + 2
```

werden wir ablehnen, obwohl es nicht zu einem Laufzeitfehler führt.

Typen

Die Typen, die wir zunächst erlauben, sind folgendermaßen definiert: Int ist ein Typ. Wenn t und u Typen sind, dann ist $t \to u$ ein Typ. Welche Typen es noch gibt, hängt von den Data-Deklarationen in einem Programm ab. Wir werden zunächst nur einfache Data-Deklarationen mit Kind $*$ betrachten, also zum Beispiel `Bool` und `IntPair`. Die Menge aller Typen bezeichnen wir mit Type.

Unser erstes Ziel ist es, formal zu definieren, welche Ausdrücke welche Typen bekommen dürfen. Dazu müssen wir ähnlich wie bei der Semantik in der Lage sein, Ausdrücken mit Variablen Typen zuzuordnen, also zum Beispiel $x + 0$. Wenn wir x mit Int belegen, dann hat $x + 0$ den Typ Int.
Eine *Typumgebung* ist eine partielle Funktion Var $\to$ Type. Wir benutzen hier dieselben Schreibweisen wie für normale Umgebungen, also $\oslash$ und $[x_1/\tau_1, \ldots, x_n/\tau_n]$.

Typurteile

Wir wollen nun formal definieren, welche Ausdrücke welche Typen bekommen können. Wir schreiben $\Gamma \vdash e : \tau$ für den Fall, dass $e \in \text{Exp}$ den Typ $\tau \in \text{Type}$ unter der Typumgebung $\Gamma \in \text{TEnv}$ haben kann. Dies nennt man auch ein *Typurteil*.

Analog zu normalen Umgebungen fordern wir, dass jede Typumgebung einen Eintrag für die fundamentalen Operationen wie $(+)$ hat, also zum Beispiel soll für jedes Γ gelten, dass $\Gamma((+)) = \text{Int} \to \text{Int} \to \text{Int}$.

Wir definieren nun $\Gamma \vdash e : \tau$ induktiv, indem wir *Typregeln* angeben. Die Basisfälle sind folgende:

Für $x \in \text{Var}$ gilt $\Gamma \vdash x : \tau$, falls $\Gamma(x) = \tau$.

Für $i \in \mathbb{Z}$ gilt $\Gamma \vdash i : \text{Int}$.

Für einen Wertkonstruktor $\underline{\text{C}}$ vom Typ τ gilt $\Gamma \vdash \underline{\text{C}} : \tau$.

Typregeln

Induktionsschritte schreiben wir als Bruch. Das, was über dem Strich steht, sind die Voraussetzungen, und das, was unter dem Strich steht, kann daraus geschlossen werden.
Für die Applikation fordern wir, dass auf der linken Seite ein passender Funktionstyp steht:

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\ e_2) : \tau}$$

Bei **let** $x = e_1$ **in** e_2 müssen wir den Typ für x sowohl in e_1, als auch in e_2 binden. Der Ergebnistyp ist dann der von e_2:

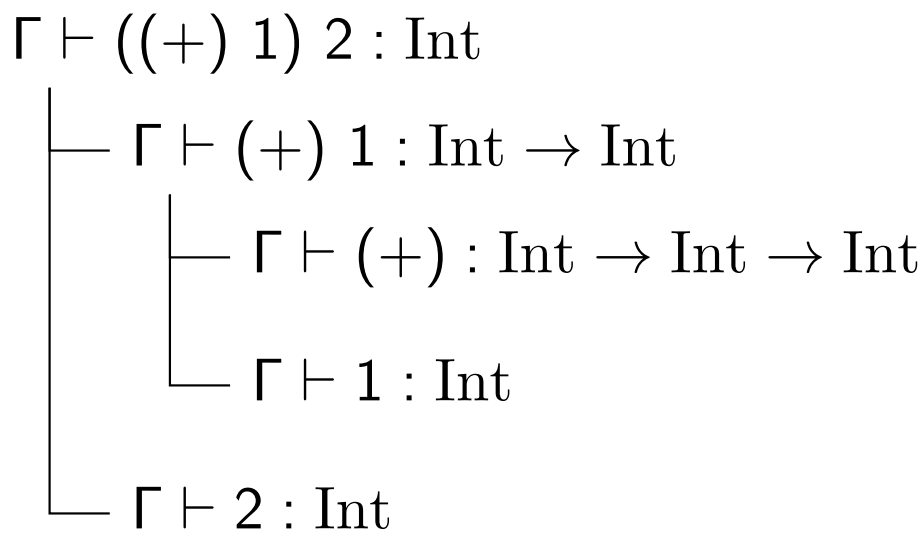
$$\frac{\Gamma \oslash [x/\tau'] \vdash e_1 : \tau' \quad \Gamma \oslash [x/\tau'] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Ein Lambda-Ausdruck erhält einen Funktionstyp bei passendem Parametertyp:

$$\frac{\Gamma \oslash [x/\tau'] \vdash e : \tau}{\Gamma \vdash \backslash x -> e : \tau' \to \tau}$$

Typregeln

Eine *Typherleitung* ist eine Folge von Typurteilen, die sich aus den Typregeln ergeben. Wir schreiben diese als Baum auf. Beispiel:



Ein Ausdruck $e \in \text{Exp}$, für den es ein $\Gamma \in \text{TEnv}$ und $\tau \in \text{Type}$ gibt mit $\Gamma \vdash e : \tau$, heißt auch *wohlgetypt*.
Nicht wohlgetypte Ausdrücke sind dann solche, für die wir keine Typherleitung finden können. Zum Beispiel ist $(+)\ 1\ \underline{\text{True}}$ nicht wohlgetypt.

Typregeln

Für Case-Ausdrücke betrachten wir folgendes Beispiel:

$$\textbf{case}\ e\ \textbf{of}\ \{\underline{\mathrm{MkPair}}\ x\ y\ {->}\ x + y\},$$

wobei $\underline{\mathrm{MkPair}}\colon \mathrm{Int} \to \mathrm{Int} \to \mathrm{IntPair}$. Wir müssen zum einen prüfen, dass $\Gamma \vdash e : \mathrm{IntPair}$, also dass der Typ von e auf den Ergebnistyp des Konstruktors passt. Zum anderen müssen wir überprüfen, dass $x + y$ unter der Umgebung $\Gamma \oslash [x/\mathrm{Int}, y/\mathrm{Int}]$ wohlgetypt ist. Hier gilt $\Gamma \oslash [x/\mathrm{Int}, y/\mathrm{Int}] \vdash x + y : \mathrm{Int}$.

Wir wollen allgemein definieren, wann Folgendes gilt:

$$\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\underline{\mathrm{C}}\ x_1\ \ \ldots\ \ x_n\ {->}\ e'\} : \tau$$

Dazu muss zunächst $\Gamma \vdash e : \tau'$ gelten. Der Ergebnistyp von $\underline{\mathrm{C}}$ muss zu τ' passen, also $\Gamma \vdash \underline{\mathrm{C}}\colon \tau_1 \to \cdots \to \tau_n \to \tau'$. Außerdem müssen die Paramtertypen von $\underline{\mathrm{C}}$ zu $x_1, \ldots, x_n$ passen, also $\Gamma \oslash [x_1/\tau_1, \ldots, x_n/\tau_n] \vdash e' : \tau$.

Typregel für Case

Bei der allgemeinen Typregel für Case, wo es mehrere Patterns geben kann, müssen wir die Forderung, die wir an einen einzelnen Pattern gestellt haben, entsprechend wiederholen:

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau' \\ \Gamma \vdash \underline{\mathrm{C}_1} : \tau_{1,1} \to \cdots \to \tau_{1,m_1} \to \tau' \\ \Gamma \oslash [x_{1,1}/\tau_{1,1}, \ldots, x_{1,m_1}/\tau_{1,m_1}] \vdash e_1 : \tau \\ \ldots \\ \Gamma \vdash \underline{\mathrm{C}_n} : \tau_{n,1} \to \cdots \to \tau_{n,m_n} \to \tau' \\ \Gamma \oslash [x_{n,1}/\tau_{n,1}, \ldots, x_{n,m_n}/\tau_{n,m_n}] \vdash e_n : \tau\end{array}}{\begin{array}{l}\textbf{case } e \textbf{ of } \{\underline{\mathrm{C}_1}\ x_{1,1}\ \ldots\ x_{1,m_1} -> e_1 \\ \qquad ; \ldots ; \\ \qquad \underline{\mathrm{C}_n}\ x_{n,1}\ \ldots\ x_{n,m_n} -> e_n\} : \tau\end{array}}$$

Beispiel für eine Typherleitung

Als Beispiel betrachten wir folgende Typherleitung für einen Ausdruck mit mehreren Patterns:

$$\Gamma \vdash \textbf{case}\ (\underline{\text{Succ}}\ \underline{\text{Zero}})\ \textbf{of}\ \{\underline{\text{Zero}} \mathrel{-\!\!>} \underline{\text{Zero}}\ ;\ \underline{\text{Succ}}\ x \mathrel{-\!\!>} x\} : \text{Nat}$$

- $\Gamma \vdash (\underline{\text{Succ}}\ \underline{\text{Zero}}) : \text{Nat}$
  - $\Gamma \vdash \underline{\text{Succ}} : \text{Nat} \to \text{Nat}$
  - $\Gamma \vdash \underline{\text{Zero}} : \text{Nat}$
- $\Gamma \vdash \underline{\text{Zero}} : \text{Nat}$
- $\Gamma \vdash \underline{\text{Zero}} : \text{Nat}$
- $\Gamma \vdash \underline{\text{Succ}} : \text{Nat} \to \text{Nat}$
- $\Gamma \circledcirc [x/\text{Nat}] \vdash x : \text{Nat}$

Typinferenz

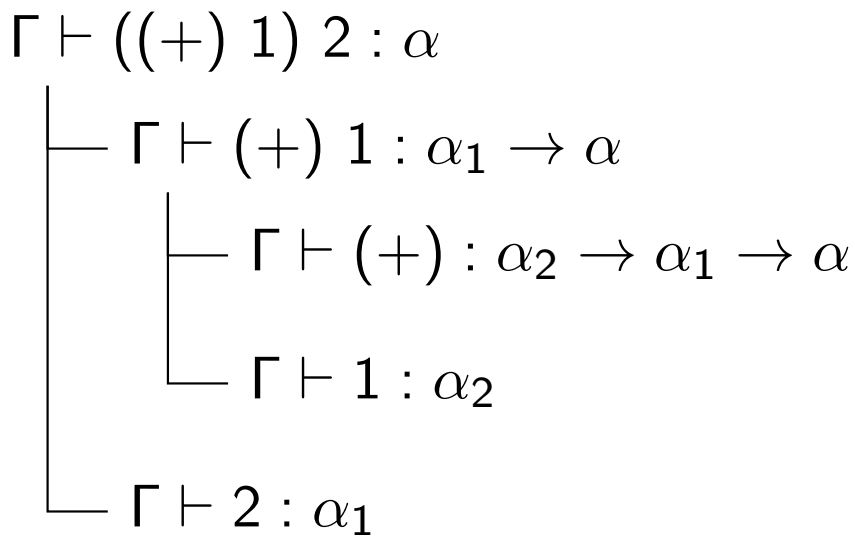
Wir werden nun einen Algorithmus implementieren, der herausfindet, ob $\Gamma \vdash e : \tau$ gilt. Die Typregeln eignen sich nicht direkt dazu, weil man an gewissen Stellen „raten“ muss, zum Beispiel bei der Applikation: Wenn man zeigen will, dass $\Gamma \vdash (e_1\ e_2) : \tau$, muss man zeigen, dass $\Gamma \vdash e_1 : \tau' \to \tau$ und $\Gamma \vdash e_2 : \tau'$, aber für welches τ'?

Wir lösen dieses Problem des „Ratens“, indem wir τ' erst einmal „offenlassen“. Dies geschieht durch Verwenden einer *Typvariable* α, also $\Gamma \vdash e_1 : \alpha \to \tau$. Formal sind die Typvariablen die Menge $\mathrm{TVar} = \{\alpha, \alpha_1, \alpha_2, \dots\}$, wobei wir TVar zu Type hinzunehmen.

Statt zu fragen, ob $\Gamma \vdash e : \tau$ für ein konkretes τ gilt, werden wir auch hier eine Typvariable verwenden, also $\Gamma \vdash e : \alpha$. Das Bestimmen des „besten“ Werts für α nennt man *Typinferenz*.

Typinferenz

Für unser erstes Beispiel sieht das dann so aus:



Unsere bisherigen Typregeln erlauben nur, dass $\Gamma \vdash (+) : \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$, und $\Gamma \vdash 1 : \mathrm{Int}$ bzw. $\Gamma \vdash 2 : \mathrm{Int}$. Statt diese Regeln zu modifizieren, notieren wir uns die Typgleichungen $\alpha_2 \to \alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$, $\alpha_2 = \mathrm{Int}$ und $\alpha_1 = \mathrm{Int}$.
Formal ist eine *Typgleichung* von der Form $\tau = \tau'$, wobei $\tau, \tau' \in$ Type, und ein *Typgleichungssystem* ist eine Menge von Typgleichungen.

Typinferenz

Unsere Aufgabe wird es sein, solch ein Typgleichungssystem zu *lösen*. Intuitiv geschieht dies, indem man Typvariablen so ersetzt, dass alle Typgleichungen von der Form $\tau = \tau$ sind, also zum Beispiel $\alpha = \mathrm{Int}$ wird zu $\mathrm{Int} = \mathrm{Int}$, indem man α durch Int ersetzt. Einen Algorithmus, der die „beste“ Lösung findet, werden wir in der nächsten Vorlesung kennenlernen.

Wenn wir eine Typgleichung $\tau = \tau'$ notieren müssen, schreiben wir dies als $\{\tau = \tau'\}$ mit in die Typregel. Die neuen Typregeln sehen dann folgendermaßen aus:

Für $x \in \mathrm{Var}$ haben wir $\Gamma \vdash x : \tau \ \{\tau = \tau'\}$, falls $\Gamma(x) = \tau'$.

Für $i \in \mathbb{Z}$ haben wir $\Gamma \vdash i : \tau \ \{\tau = \mathrm{Int}\}$.

Für einen Wertkonstruktor $\underline{\mathrm{C}}$ vom Typ τ' haben wir $\Gamma \vdash \underline{\mathrm{C}} : \tau \ \{\tau = \tau'\}$.

Typinferenz

Bei der Applikation ersetzen wir, wie besprochen, den Parameter-Typ durch eine neue Typvariable:

$$\frac{\Gamma \vdash e_1 : \alpha \to \tau \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1\ e_2) : \tau}$$

Bei Let-Ausdrücken gehen wir analog vor:

$$\frac{\Gamma \oslash [x/\alpha] \vdash e_1 : \alpha \quad \Gamma \oslash [x/\alpha] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Bei Lambda-Ausdrücken können wir nicht mehr fordern, dass der Ergebnistyp ein Funktionstyp ist. Daher fügen wir zwei neue Typvariablen (für den Parameter und für das Ergebnis) ein:

$$\frac{\Gamma \oslash [x/\alpha] \vdash e : \alpha'}{\Gamma \vdash \backslash x \texttt{->}\ e : \tau\ \{\tau = \alpha \to \alpha'\}}$$

Beispiel für Typinferenz

Für unser vorheriges Beispiel sähe eine Typinferenz dann so aus:

$$\Gamma \vdash ((+)\ 1)\ 2 : \alpha$$
$$\quad \Gamma \vdash (+)\ 1 : \alpha_1 \to \alpha$$
$$\quad\quad \Gamma \vdash (+) : \alpha_2 \to \alpha_1 \to \alpha$$
$$\quad\quad \{\alpha_2 \to \alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}\}$$
$$\quad\quad \Gamma \vdash 1 : \alpha_2\ \{\alpha_2 = \mathrm{Int}\}$$
$$\quad \Gamma \vdash 2 : \alpha_1\ \{\alpha_1 = \mathrm{Int}\}$$

Damit erhalten wir folgendes Typgleichungssystem:

$$\begin{aligned} \{\alpha_2 \to \alpha_1 \to \alpha &= \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}, \\ \alpha_2 &= \mathrm{Int}, \\ \alpha_1 &= \mathrm{Int}\} \end{aligned}$$

Typinferenz

Bei Case-Ausdrücken ersetzen wir einfach alle Typen, die wir bisher „raten“ mussten, durch neue Typvariablen:

$$\frac{\begin{array}{l}\Gamma \vdash e : \alpha \\ \Gamma \vdash \underline{\mathrm{C}_1} : \alpha_{1,1} \to \cdots \to \alpha_{1,m_1} \to \alpha \\ \Gamma \oslash [x_{1,1}/\alpha_{1,1}, \ldots, x_{1,m_1}/\alpha_{1,m_1}] \vdash e_1 : \tau \\ \ldots \\ \Gamma \vdash \underline{\mathrm{C}_n} : \alpha_{n,1} \to \cdots \to \alpha_{n,m_n} \to \alpha \\ \Gamma \oslash [x_{n,1}/\alpha_{n,1}, \ldots, x_{n,m_n}/\alpha_{n,m_n}] \vdash e_n : \tau \end{array}}{\begin{array}{l}\textbf{case } e \textbf{ of } \{\underline{\mathrm{C}_1}\ x_{1,1}\ \ldots\ x_{1,m_1} \mathrel{->} e_1 \\ \qquad ; \ldots ; \\ \qquad \underline{\mathrm{C}_n}\ x_{n,1}\ \ldots\ x_{n,m_n} \mathrel{->} e_n\} : \tau\end{array}}$$

Weiteres Beispiel für Typinferenz

$\Gamma \vdash \textbf{case}\ (\underline{\text{Succ}}\ \underline{\text{Zero}})\ \textbf{of}\ \{\underline{\text{Zero}} \mathrel{->} \underline{\text{Zero}}\ ;\ \underline{\text{Succ}}\ x \mathrel{->} x\} : \alpha$

- $\Gamma \vdash (\underline{\text{Succ}}\ \underline{\text{Zero}}) : \alpha_1$
  - $\Gamma \vdash \underline{\text{Succ}} : \alpha_2 \to \alpha_1\ \{\alpha_2 \to \alpha_1 = \text{Nat} \to \text{Nat}\}$
  - $\Gamma \vdash \underline{\text{Zero}} : \alpha_2\ \{\alpha_2 = \text{Nat}\}$
- $\Gamma \vdash \underline{\text{Zero}} : \alpha_1\ \{\alpha_1 = \text{Nat}\}$
- $\Gamma \vdash \underline{\text{Zero}} : \alpha\ \{\alpha = \text{Nat}\}$
- $\Gamma \vdash \underline{\text{Succ}} : \alpha_3 \to \alpha_1\ \{\alpha_3 \to \alpha_1 = \text{Nat} \to \text{Nat}\}$
- $\Gamma \oslash [x/\alpha_3] \vdash x : \alpha\ \{\alpha_3 = \alpha\}$

Wir erhalten damit das Typgleichungssystem:

$$\{\alpha_2 \to \alpha_1 = \text{Nat} \to \text{Nat},\ \alpha_2 = \text{Nat},\ \alpha_1 = \text{Nat},$$
$$\alpha = \text{Nat},\ \alpha_3 \to \alpha_1 = \text{Nat} \to \text{Nat},\ \alpha_3 = \alpha\}$$

Unifikation

Wir wollen im Folgenden Typgleichungssysteme lösen. Zur Erinnerung: Wir lösen diese, indem wir Typvariablen ersetzen. Eine *Substitution* ist eine partielle Funktion TVar $\to$ Type. Analog zu Umgebungen schreiben wir $[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$ für die Substitution s mit $s(\alpha_i) = \tau_i$ für $1 \leq i \leq n$.
Das Anwenden einer Substitution s auf einen Typ $\tau \in$ Type schreiben wir als τs, wobei

$$\tau s = \begin{cases} \tau_1 s \to \tau_2 s & \text{falls } \tau = \tau_1 \to \tau_2, \\ s(\alpha) & \text{falls } \tau = \alpha \text{ und } s(\alpha) \text{ definiert,} \\ \tau & \text{sonst.} \end{cases}$$

Das heißt, für jede Typvariable α, für die s einen Eintrag hat, ersetzen wir jedes Vorkommen von α durch $s(\alpha)$.
Beispiel: $\alpha \to \alpha_1[\alpha/\mathrm{Int}] = \mathrm{Int} \to \alpha_1$.

Unifikation

Analog können wir eine Substitution s auf Typgleichungen und auch auf Typgleichungssysteme anwenden, was wir als $(\tau = \tau')s$ bzw. Es für ein Typgleichungssystem E schreiben.

Die *Erweiterung* einer Substitution s_1 durch eine Substitution s_2 ist die Substitution $s_1 s_2$, wobei $(s_1 s_2)(\alpha) = (s_1(\alpha))s_2$. Das heißt, die Einträge von s_2 „wirken hinterher“ auf die von s_1. Beispiel: Sei $s_1 = [\alpha/\alpha_1]$ und $s_2 = [\alpha_1/\mathrm{Int}]$. Dann ist $s_1 s_2 = [\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}]$ und $s_2 s_1 = [\alpha_1/\mathrm{Int}, \alpha/\alpha_1]$.

Eine Substitution s heißt *Lösung* einer Typgleichung $\tau = \tau'$ genau dann, wenn τs und $\tau s'$ gleich sind. Entsprechend heißt s Lösung eines Typgleichungssystems, wenn es Lösung aller Typgleichungen in ihm ist.

Unifikation

Beim Lösen eines Typgleichungssystem sind wir an der „besten“ Lösung interessiert. Eine Substitution s_1 ist *allgemeiner als* eine Substitution s_2, wenn es eine Substitution s_3 gibt mit $s_2 = s_1 s_3$. Das heißt, man erhält aus dem „allgemeineren“ s_1 ein „spezielleres“ s_2, indem man weitere Ersetzungen, nämlich die aus s_3, durchführt. Beispiel: $s_1 = [\alpha/\alpha_1]$ ist allgemeiner als $s_2 = [\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}]$, weil $s_2 = s_1[\alpha_1/\mathrm{Int}]$. Eine Lösung heißt *allgemeinste Lösung*, wenn sie allgemeiner als alle anderen Lösungen ist.

Beispiel: Sei $E = \{\alpha \to \mathrm{Int} = \alpha_1 \to \alpha_2\}$. Eine allgemeinste Lösung für E wäre $s_1 = [\alpha/\alpha_1, \alpha_2/\mathrm{Int}]$, also $Es_1 = \{\alpha_1 \to \mathrm{Int} = \alpha_1 \to \mathrm{Int}\}$. Eine andere Lösung wäre $[\alpha/\mathrm{Int}, \alpha_1/\mathrm{Int}, \alpha_2/\mathrm{Int}]$, also $Es_2 = \{\mathrm{Int} \to \mathrm{Int} = \mathrm{Int} \to \mathrm{Int}\}$.

Die *freien Variablen* eines Typs sind definiert als $\mathrm{free}(\alpha) = \{\alpha\}$ für $\alpha \in \mathrm{TVar}$, $\mathrm{free}(\mathrm{Int}) = \emptyset$ und $\mathrm{free}(\tau \to \tau') = \mathrm{free}(\tau) \cup \mathrm{free}(\tau')$.

Unifikationsalgorithmus

Wir geben nun einen Algorithmus an, der ein Typgleichungssystem E bekommt und

- „nicht lösbar“ liefert, wenn es keine Lösung für E gibt,
- und andernfalls eine allgemeinste Lösung für E liefert.

Wir starten mit einer Substitution s, die am Anfang leer ist und verwalten ein Typgleichungssystem E', das zu Anfang auf E gesetzt wird. In jedem Schritt wählen wir aus E' eine Gleichung $\tau = \tau'$ aus, die wir auch aus E' entfernen. Wir sind fertig, wenn E' leer ist. Dann ist s die allgemeinste Lösung.

Im Fall, dass die aktuelle Typgleichung schon gelöst ist, also die Form $\tau = \tau$ hat, ist nichts weiter zu tun.

Unifikationsalgorithmus

Im Fall, dass die aktuelle Typgleichung $\tau = \alpha$ ist, wobei $\alpha \in \text{TVar}$, schauen wir, ob α in τ vorkommt, also $\alpha \in \text{free}(\tau)$. Wenn ja, ist das Ergebnis „nicht lösbar“. Beispiel: $\alpha = \alpha \to \text{Int}$ hat keine Lösung. Ansonsten setzen wir s auf $s[\alpha/\tau]$. Außerdem müssen wir $[\alpha/\tau]$ auf E' anwenden. Der Fall, dass die Typgleichung die Form $\alpha = \tau$ hat, ist analog.

Eine Typgleichung der Form $\tau_1 \to \tau_1' = \tau_2 \to \tau_2'$ lösen wir, indem wir die Typgleichungen $\tau_1 = \tau_2$ und $\tau_1' = \tau_2'$ zu E' hinzufügen. Wir ersetzen also eine Typgleichung durch zwei. Beispiel: $\text{Int} \to \alpha = \alpha_1 \to \text{Bool}$ ersetzen wir durch $\text{Int} = \alpha_1$ und $\alpha = \text{Bool}$.

Falls wir auf eine Gleichung $\tau = \tau'$ stoßen, die nicht auf die vorherigen Fälle passt, ist das Ergebnis „nicht lösbar“. Beispiel: $\text{Int} = \alpha \to \text{Bool}$.

Typinferenz

Ein Typ τ heißt *allgemeiner* als ein Typ τ', wenn es eine Substitution s gibt mit $\tau s = \tau'$. Beispiel: $\alpha \to \mathrm{Int}[\alpha/\mathrm{Int}] = \mathrm{Int} \to \mathrm{Int}$. Also ist $\alpha \to \mathrm{Int}$ allgemeiner als $\mathrm{Int} \to \mathrm{Int}$. Zu einem Ausdruck e heißt τ *allgemeinster Typ* von e, wenn $\Gamma \vdash e : \tau$ und für alle anderen Typen τ' mit $\Gamma \vdash e : \tau'$ gilt, dass τ allgemeiner ist als τ'.

Insgesamt liefert uns Typinferenz den allgemeinsten Typ zu einem Ausdruck e, wenn dieser wohlgetypt ist. Wir starten mit $\Gamma \vdash e : \alpha$, wobei $\alpha \in \mathrm{TVar}$ und erhalten daraus ein Gleichungssystem E. Darauf wenden wir den Unifikationsalgorithmus an. Dieser liefert „nicht lösbar“, wenn E keine Lösung hat. In dem Fall ist e nicht wohlgetypt. Wenn E lösbar ist, dann liefert der Unifikationsalgorithmus eine allgemeinste Lösung s von E. In dem Fall ist $s(\alpha)$ der allgemeinste Typ von e.

Beispiel für Unifikation

In der letzten Vorlesung haben wir angefangen, Typinferenz für $\Gamma \vdash ((+)\ 1)\ 2 : \alpha$ durchzuführen. Dabei haben wir folgendes Typgleichungssystem erhalten:

$$E := \{\alpha_2 \to (\alpha_1 \to \alpha) = \mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int}),\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Wir zeigen nun, wie der Unifikationsalgorithmus dafür durchgeführt werden kann. Wir starten mit $E' := E$ und $s = []$.

Wir wählen zunächst $\alpha_2 \to (\alpha_1 \to \alpha) = \mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int})$ aus. Dies müssen wir durch $\alpha_2 = \mathrm{Int}$ und $\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int}$ ersetzen und erhalten

$$E' := \{\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int},\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Dann wählen wir $\alpha_1 \to \alpha = \mathrm{Int} \to \mathrm{Int}$ und erhalten

$$E' := \{\alpha = \mathrm{Int},\ \alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Beispiel für Unifikation

Wir wählen $\alpha = \mathrm{Int}$ und erhalten $s := s[\alpha/\mathrm{Int}] = [\alpha/\mathrm{Int}]$ mit

$$E' := \{\alpha_2 = \mathrm{Int},\ \alpha_1 = \mathrm{Int}\}.$$

Dann wählen wir $\alpha_2 = \mathrm{Int}$ und erhalten
$s := s[\alpha_2/\mathrm{Int}] = [\alpha/\mathrm{Int}, \alpha_2/\mathrm{Int}]$ mit

$$E' := \{\alpha_1 = \mathrm{Int}\}.$$

Zuletzt wählen wir $\alpha_1 = \mathrm{Int}$ und erhalten
$s := s[\alpha_1/\mathrm{Int}] = [\alpha/\mathrm{Int}, \alpha_2/\mathrm{Int}, \alpha_1/\mathrm{Int}]$ mit $E' := \emptyset$.

Damit ist der Algorithmus fertig und wir haben die allgemeinste Lösung gefunden. Wegen $s(\alpha) = \mathrm{Int}$ ist Int der allgemeinste Typ von $((+)\ 1)\ 2$.

Polymorphie

Eine Funktion wie `f x = x` nennt man polymorph, weil man sie mit allen Typen benutzen kann. In Haskell schreiben wir `f :: a -> a`, was *nicht* gleich $\Gamma \vdash f : \alpha \to \alpha$ ist!
Zum Beispiel bekommen wir bei der Typinferenz für

```
let f = \x -> x
in case f True of { True -> f 0 }
```

heraus, dass $\alpha = \mathrm{Int}$ und $\alpha = \mathrm{Bool}$, also $\mathrm{Int} = \mathrm{Bool}$ (Übung).

In Haskell ist `f :: a -> a` hingegen eine Abkürzung für `f :: forall a. a -> a`. Um das in unserem Typsystem abzubilden, definieren wir die neuen Typen

$$\mathsf{PType} = \{\forall\alpha_1 \cdots \forall\alpha_n.\tau \mid \tau \in \mathsf{Type}\,,\ \alpha_1, \ldots, \alpha_n \in \mathsf{TVar}, n \geq 0\}.$$

Man beachte, dass $\mathsf{Type} \subseteq \mathsf{PType}$.

Polymorphie

Wir werden bei **let** $f = e_1$ **in** e_2 erlauben, dass f einen polymorphen Typ in e_2 bekommt. In unserem vorherigen Beispiel würde also für f der Typ $\forall\alpha.\alpha \to \alpha$ in Γ eingetragen, sodass f sowohl auf $\underline{\mathrm{True}}$, als auch auf 0 angewandt werden kann.
In e_1 hingegen, also der Definition von f, darf f keinen polymorphen Typ bekommen.
Wichtig ist außerdem, dass nur die Typvariablen ein $\forall$ in e_2 bekommen dürfen, die nicht schon in Γ vorkommen. Man könnte zum Beispiel die Identität folgendermaßen definieren:

$$\textbf{let}\ \mathrm{id} = \backslash x \text{->} (\textbf{let}\ g = \backslash y \text{->} x\ \textbf{in}\ g\ 0)\ \textbf{in}\ e$$

Könnte man einfach $\forall$ vor alle Typvariablen schreiben, könnte man g den Typ $\forall\alpha_1\forall\alpha_2.\alpha_1 \to \alpha_2$ in $g\ 0$ geben, was wiederum dazu führt, dass auch id diesen Typ in e bekommt.

Polymorphie

Für Typkonstruktoren mit Kind

$$T : \underbrace{* \to \cdots \to *}_{n \text{ mal}} \to *$$

müssen wir unsere induktive Definition von Typen erweitern: Wenn $\tau_1, \ldots, \tau_n \in \text{Type}$, dann $T(\tau_1, \ldots, \tau_n) \in \text{Type}$. Zum Beispiel sind $\mathrm{List}(\alpha)$ und $\mathrm{List}(\mathrm{Int})$ Typen. Entsprechend müssen Substitutionen s auch in solchen Typen die Typvariablen ersetzen, also

$$T(\tau_1, \ldots, \tau_n)s = T(\tau_1 s, \ldots, \tau_n s).$$

Zum Beispiel ist $\mathrm{List}(\alpha)[\alpha/\mathrm{Int}] = \mathrm{List}(\mathrm{Int})$.

Ebenso definieren wir free als

$$\mathsf{free}(T(\tau_1, \ldots, \tau_n)) = \mathsf{free}(\tau_1) \cup \cdots \cup \mathsf{free}(\tau_n).$$

Polymorphie

Typumgebungen können nun polymorphe Typen enthalten, also sind diese jetzt partielle Funktionen TVar $\to$ PType. Typurteile sind immer noch von der Form $\Gamma \vdash e : \tau$, wobei $\tau \in$ Type!

Die *freien Variablen* in polymorphen Typen seien definiert als $\mathsf{free}(\forall\alpha_1 \ldots \forall\alpha_n.\tau) = \mathsf{free}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$. Zum Beispiel ist $\mathsf{free}(\forall\alpha.\alpha \to \alpha_1) = \{\alpha_1\}$.

Für eine Typumgebung Γ sei

$$\mathsf{free}(\Gamma) = \bigcup\{\mathsf{free}(\Gamma(x)) \mid x \in \mathsf{Var}\,,\ \Gamma(x) \text{ definiert}\}.$$

Zum Beispiel ist $\mathsf{free}([x/\forall\alpha.\alpha, y/\mathrm{Int} \to \alpha_1]) = \{\alpha_1\}$.

Der *polymorphe Abschluss* $\mathsf{cl}_\Gamma(\tau)$ für $\Gamma \in$ TEnv und $\tau \in$ Type ist $\forall\alpha_1 \cdots \forall\alpha_n.\tau$, wobei $\{\alpha_1, \ldots, \alpha_n\} = \mathsf{free}(\tau) \setminus \mathsf{free}(\Gamma)$.
Zum Beispiel gilt $\mathsf{cl}_{[]}(\alpha \to \alpha_1 \to \mathrm{Int}) = \forall\alpha\forall\alpha_1.\alpha \to \alpha_1 \to \mathrm{Int}$, aber $\mathsf{cl}_{[x/\alpha]}(\alpha \to \alpha_1 \to \mathrm{Int}) = \forall\alpha_1.\alpha \to \alpha_1 \to \mathrm{Int}$.

Polymorphie

Wir passen nun unsere Regeln an: Für $x \in \text{Var}$ erlauben wir, einen polymorphen Typ zu instanziieren, also
$\Gamma \vdash x : \tau[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$, falls $\Gamma(x) = \forall\alpha_1 \cdots \forall\alpha_n.\tau$. Beispiel:
$[f/\forall\alpha.\alpha \to \text{Int}] \vdash f : \text{Bool} \to \text{Int}$.

Ebenso erlauben wir dies für Wertkonstruktoren, also
$\Gamma \vdash \underline{\text{C}} : \tau[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$, falls $\underline{\text{C}} : \forall\alpha_1 \cdots \forall\alpha_n.\tau$
Beispiel: Es gilt $\underline{\text{Nil}} : \forall\alpha.\text{List}(\alpha)$, also auch $\Gamma \vdash \underline{\text{Nil}} : \text{List}(\text{Int})$.

Die Typregel für Let ändern wir wie besprochen:

$$\frac{\Gamma \oslash [x/\tau'] \vdash e_1 : \tau' \quad \Gamma \oslash [x/\,\text{cl}_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Auch für das polymorphe Typsystem lässt sich Typinferenz implementieren, was wir allerdings nicht mehr behandeln können.

Beispiel für Polymorphie

$\Gamma \vdash \textbf{let } f = \backslash x \text{ -> } x \textbf{ in case } f \ \underline{\text{True}} \textbf{ of } \{\underline{\text{True}} \text{ -> } f \ 0\} : \text{Int}$

- $\Gamma \circledcirc [f/\alpha \to \alpha] \vdash \backslash x \text{ -> } x : \alpha \to \alpha$
  - $\Gamma \circledcirc [f/\alpha \to \alpha, x/\alpha] \vdash x : \alpha$
- $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \textbf{case } f \ \underline{\text{True}} \textbf{ of } \{\underline{\text{True}} \text{ -> } f \ 0\} : \text{Int}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f \ \underline{\text{True}} : \text{Bool}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f : \text{Bool} \to \text{Bool}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \underline{\text{True}} : \text{Bool}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash \underline{\text{True}} : \text{Bool}$
  - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f \ 0 : \text{Int}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash f : \text{Int} \to \text{Int}$
    - $\Gamma \circledcirc [f/\forall\alpha.\alpha \to \alpha] \vdash 0 : \text{Int}$