Skript Programmiersprachen 1

David Läwen

November 5, 2020

GitHub-Repository mit Code und weiteren Materialien:
https://github.com/davidlaewen/programmiersprachen1

CodiMD-Version mit schönerer Formatierung:
https://pad.its-amazing.de/programmiersprachen1

Contents

Vorlesungsinhalte

- Verständnis von Programmiersprachen (allgemein, über aktuelle "Trends" hinweg) und deren Qualitäten, Vor- und Nachteile
- Zerlegung von Programmiersprachen in Features sowie deren Analyse
- Erlernen von strukturiertem Programmdesign durch Abstraktionen
- Befähigung, auf informierte Art Programmiersprachen einzuschätzen und zu diskutieren
- Implementieren von Programmiersprachen(-features) durch Interpreter in Scala
- Zweck/Nutzen verschiedener Sprachfeatures, mögliche Implementationen und deren Vorzüge/Probleme

Scala-Grundlagen

Scala ist statisch getypt, funktional und objekt-orientiert. Scala-Programme werden zu Java-Bytecode kompiliert und in der JVM ausgeführt. Auswertung ist *eager* (*Call By Value*). - **Konstanten** werden mit `val` und mutierbare **Variablen** mit `var` definiert. Der Typ muss dabei nicht deklariert werden (wird aber dennoch von Scala festgelegt), also bspw. `var n = 1` oder `var s = "abc"`. Der Typ kann aber auch explizit deklariert werden, also bspw. `var n: Int = 1` oder `var s: String = "abc"`.

- **Funktionen** haben die Form `def f(param: Type, ...) = body` und werden durch `f(arg,...)` aufgerufen. Der Rückgabetyp kann optional angegeben werden: `def f(param: Type, ...): ReturnType = body`

```
def square(n: Int) : Int = n*n
val x = square(4)

def concat(a: String, b: String): String = a + b
val y = concat("Heiner", concat(" ", "Hacker"))
```

Datentypen

- Es gibt die gängigen Datentypen `Int`, `String`, `Boolean`, `Double`, etc.
- `Unit` entspricht dem Rückgabetyp `void` in Java (kein Rückgabewert, sondern "Seiteneffekt", bspw. bei der `print`-Funktion)
- **Map:** Abbildung mit Key-Value-Paaren.

```
var map: Map[Int,String] = Map(1 -> "a", 2 -> "b")
assert(map(1) == "a")
```

- **Tupel:** Erlauben Gruppierung heterogener Daten. Können zwischen 2 und 22 Werte beliebigen Typs enthalten.

```
val t: (Int,String,Boolean) = (1, "abc", true)
val firstEntry = t._1
val (int, string, boolean) = t // binds all three fields
assert(firstEntry == int)
```

- **Listen:** Besitzen einen einheitlichen Typ, bspw. `List[Int]`, Einträge sind nicht mutierbar.

```
val nums = List(1,2,3)
assert(nums(0) == 1)

val nums = List.range(0, 10)
val nums = (1 to 10 by 2).toList
val letters = ('a' to 'f' by 2).toList

letters.foreach(println)
nums.filter(_ > 3).foreach(println)
val doubleNums = nums.map(_ * 2)
val bools = nums.map(_ < 5)
val squares = nums.map(n => n*n)
val sum = nums.fold(0)(_+_)
val prod = nums.fold(1)(_*_)
```

- **Arrays:** Besitzen auch einheitlichen Typ, sind im Gegensatz zu Listen mutierbar.

```
val a = Array(1,2,3)
assert(a(0) == 1)
a(0) = 10
assert(a(0) == 10)
```

- **Mengen:** `Set(1,2,3)`, mutierbar mit `+` und `-` um einzelne Werte hinzuzufügen bzw. zu entfernen, `++` und `--` für Vereinigung bzw. Schnitt mit anderem `Collectible`-Datentyp, bspw. Menge oder Liste.
- **Either:** Repräsentiert einen Wert eines von zwei möglichen Typen. Jede Instanz von `Either` ist entweder eine Instanz von `Left` oder von `Right`.

```
val a: Either[Boolean,Int] = Left(true)
val b: Either[Boolean,Int] = Right(3)
```

Objektorientierung

- **Klassen:**

```
class Person(var firstName: String, var lastName: String) {
  def sayHello() = print(s"Hello, $firstName $lastName")
}

val heiner = new Person("Heiner", "Hacker")

heiner.sayHello() // prints "Hello, Heiner Hacker"
println(heiner.firstName) // prints "Heiner"
heiner.firstName = "Heinrich" // field access without get and set methods
heiner.lastName = "Knacker"
heiner.sayHello() // now prints "Hello, Heinrich Knacker"
```

- Mit `var` definierte Felder einer Klasse sind mutierbar.
- **Abstrakte Klassen** können mit dem Keyword `abstract` angelegt werden.
- **Traits** sind Bausteine zur Konstruktion von Klassen und können nicht instanziiert werden. Sie lassen sich einer Klasse mit `with`/`extends` anfügen.
- Ist ein Trait oder eine abstrakte Klasse *sealed* (Keyword `sealed ...`), so müssen alle erbenden Klassen in der gleichen Datei definiert sein. Dadurch kann bei Pattern Matching erkannt werden, ob alle Fälle (also alle Case Classes) abgedeckt sind.

```
abstract class Speaker {
    def sayCatchPhrase(): Unit // no function body, abstract
}

trait Sleeper {
    def sleep(): Unit = println("I'm sleeping")
    def wakeUp(): Unit = println("I'm awake")
}

class Person(var name: String, catchPhrase: String) extends Speaker with Sleeper {
    def sayCatchPhrase(): Unit = println(catchPhrase)
    override def sleep() = println("Zzzzzzz")
}
```

Umdefinieren von Methoden in Unterklassen ist mit dem Keyword `override` möglich. - **Objekte:** Können mit Keyword `object` instanziiert werden.

- Mit `extends` können Klassen/Traits erweitert oder abstrakte Klassen implementiert werden
- **Case Classes** sind hilfreich bei der Verwendung von Klassen als Datencontainer. Erzeugung von Instanzen ist ohne `new` möglich, zudem gibt es Default-Implementationen für das Vergleichen oder Hashen von Instanzen der Klasse. Mit Case Classes ist Pattern Matching möglich:

```
sealed abstract class UniPerson
case class Student(val id: Int) extends UniPerson
case class Professor(val subject: String) extends UniPerson

def display(p: UniPerson) : String = p match {
  case Student(id) => s"Student number $id"
  case Professor(subject) => s"Professor of $subject"
}
```

- Eine Implementation im objektorientierten Stil sieht dagegen folgendermaßen aus:

```
abstract class UniPerson {
  def display : String
}
class Student(val id: Int) extends UniPerson {
  def display = s"Student number $id"
}
class Professor(val subject: String) extends UniPerson {
  def display = s"Professor of $subject"
}
```

Die erste Variante (Pattern-Match-Dekomposition) erlaubt das Hinzufügen weiterer Funktionen, die auf dem Datentyp `UniPerson` operieren, ohne Modifikation des bestehenden Codes.

Die zweite Variante (objektorientierte Dekomposition) erlaubt das Hinzufügen weiterer Unterklassen von `UniPerson` ohne Modifikation des bestehenden Codes.

Die Schwierigkeit, die Vorzüge beider Repräsentationen zu vereinen, wird *Expression Problem* genannt (mehr dazu im Kapitel Objekt-Algebren).

Kontrollstrukturen

- *If-Else*-Statements:

```
if (1 < 2) print("Condition met")

if (a > b) {
    print("a greater than b")
} else if (a == b) {
    print("a equals b")
} else {
    print("a less than b")
}

val x = if (1 == 1) "a" else "b" // usable as ternary operator, "a" is bound to x
```

- *For-Comprehensions*:

```
for (elem <- list) println(elem)

for (i <- 0 to 10 by 2) println(i)

val evenNums = for {
  i <- 0 to 10
  if i % 2 == 0
} yield i
// evenNums: Vector(0,2,4,6,8,10)
```

Pattern Matching

```
def pm(x: Any) = x match {
  case 1 => "x is 1"
  case true => "x is true"
  case s: String => s"x is string $s"
  case (a,b,c) => s"x is tuple of $a, $b and $c"
  case (a,b) if (a == b) => s"tuple ($a,$b) and $a == $b"
  case _ => "x is something else"
}
```

REPL

- Anleitung zur Installation der Binaries hier
- REPL lässt sich mit Befehl `scala` starten
- `.scala`-Dateien lassen sich in REPL laden mit `:load filename.scala`.
- Ergebnisse werden automatisch an Variablennamen gebunden.
- Typ eines Werts kann mit `:type` abgefragt werden
- Bisherige Definitionen können mit `:reset` gelöscht werden.
- REPL kann mit `:q` verlassen werden.

Implizite Konvertierung

Scala bietet die Möglichkeit, bestimmte Typkonvertierungsfunktionen automatisch zu nutzen, wenn dadurch der erwartete Typ erfüllt werden kann. Mit dieser *impliziten Konvertierung* können wir Ausdrücke für unsere

Interpreter geschickter notieren.

Hierzu muss die `implicitConversions`-Bibliothek importiert werden:

```
import scala.language.implicitConversions

abstract class BTree
case class Node(l: BTree, r: BTree) extends BTree
case class Leaf(n: Int) extends BTree

implicit def int2btree(n: Int) : BTree = Leaf(n)

val test = Node(1,2) // is implicitly converted to Node(Leaf(1), Leaf(2))
```

Die Funktion `int2btree` wird durch das Keyword `implicit` automatisch auf Werte vom Typ `Int` aufgerufen, wenn an deren Stelle ein Wert vom Typ `BTree` erwartet wird.

Typ-Alias

Mit dem Keyword `type` können neue Typen definiert werden:

```
type IntStringMap = Map[Int, String]
```

Lambda-Ausdrücke und Currying

Es können in Scala anonyme Funktionen als Werte (*Lambda-Ausdrücke*) definiert werden. Diese haben dann einen Typ der Form `Type => ...`:

```
val succ : (n: Int) => n+1
```

Funktionen bzw. Lambda-Ausdrücke können dadurch der Rückgabewert von Funktionen sein (*Higher Order*), wodurch etwa *Currying* möglich wird:

```
def curryAdd(n: Int) : (Int => Int) = x => x+n
assert(curryAdd(3)(4) == 7)
```

Erster Interpreter (AE)

```
sealed trait Exp
case class Num(n: Int) extends Exp
case class Add(l: Exp, r: Exp) extends Exp
case class Mul(l: Exp, r: Exp) extends Exp

def eval(e: Exp) : Int = e match {
  case Num(n) => n
  case Add(l,r) => eval(l) + eval(r)
  case Mul(l,r) => eval(l) * eval(r)
}

// example expressions
var onePlusTwo = Add(Num(1), Num(2))
assert(eval(onePlusTwo) == 3)
var twoTimesFour = Mul(Num(2), Add(Num(1), Num(3)))
assert(eval(twoTimesFour) == 8)
var threeTimesFourPlusFour = Add(Mul(Num(3),Num(4)), Num(4))
assert(eval(threeTimesFourPlusFour) == 16)
```

Bei der Implementation eines Interpreters ist ein umfassendes Verständnis der *Metasprache* (hier Scala) notwendig, um die Eigenschaften der (von uns definierten) *Objektsprache* vollständig zu kennen. Bspw. betreffen die Eigenschaften und Einschränkungen des `Int`-Datentyps durch dessen Verwendung auch die Objektsprache.

Syntaktischer Zucker und Desugaring

In vielen Programmiersprachen gibt es Syntaxerweiterungen, die Programme lesbarer machen oder verkürzen, aber gleichbedeutend mit einer ausführlicheren Schreibweise sind (*syntaktischer Zucker*). Dadurch wird das Programmieren angenehmer und die Lesbarkeit von Programmen besser, für die Implementierung der Sprache ist syntaktischer Zucker jedoch lästig, da man gleichbedeutende Syntax mehrfach implementieren muss.

Der syntaktische Zucker erweitert den Funktionsumfang der Sprache nicht und jeder Ausdruck kann mit der gleichen Bedeutung ohne syntaktischen Zucker formuliert werden. Deshalb werden Sprachen typischerweise in eine *Kernsprache* und eine *erweiterte Sprache* aufgeteilt. Dann können Ausdrücke vor dem Interpretieren vollständig in die Kernsprache übersetzt werden (*Desugaring*) und der Interpreter muss nur Ausdrücke in der Kernsprache auswerten können.

Interpreter mit Desugaring

```
// core language
sealed trait CExp
case class CNum(n: Int) extends CExp
case class CAdd(l: CExp, r: CExp) extends CExp
case class CMul(l: CExp, r: CExp) extends CExp

// extended language
sealed trait SExp
case class SNum(n: Int) extends SExp
case class SAdd(l: SExp, r: SExp) extends SExp
case class SMul(l: SExp, r: SExp) extends SExp
case class SNeg(e: SExp) extends SExp
case class SSub(l: SExp, r: SExp) extends SExp

def desugar(s: SExp) : CExp = s match {
  case SNum(n) => CNum(n)
  case SAdd(l,r) => CAdd(desugar(l), desugar(r))
  case SMul(l,r) => CMul(desugar(l), desugar(r))
  case SNeg(e)   => CMul(CNum(-1), desugar(e))
  case SSub(l,r) => CAdd(desugar(l), desugar(SNeg(r)))
}

def eval(c: CExp) : Int = c match {
  case CNum(n) => n
  case CAdd(l,r) => eval(l) + eval(r)
  case CMul(l,r) => eval(l) * eval(r)
}

// examples:
var twoMinusOne = SSub(SNum(2), SNum(1))
assert(eval(desugar(twoMinusOne)) == 1)
var fivePlusFour = SAdd(SNum(5), SNum(4))
assert(eval(desugar(fivePlusFour)) == 9)
```

Für unsere Zwecke reicht es aus, syntaktischen Zucker in einer Funktion zu definieren. Dadurch ist kein Desugaring notwendig, wir können aber trotzdem Testausdrücke einfacher notieren.

```
def neg(e: Exp) = Mul(Num(-1), e)
def sub(l: Exp, r: Exp) = Add(l, neg(r))
```

Syntaktischer Zucker kann (wie bei `SSub` und `sub`) auch auf anderem syntaktischen Zucker aufbauen.

Identifier mit Umgebung (AEId)

Wir wollen unseren ersten Interpreter für arithmetische Ausdrücke um Konstantendefinitionen erweitern. Um Identifier in den Ausdrücken verwenden zu können, legen wir eine zusätzliche Datenstruktur an, nämlich eine Umgebung (*Environment*), in der Paare aus Bezeichnern und Werten hinterlegt werden.

Wir verwenden für die Identifier den Datentyp `String`, für die Umgebung definieren wir das Typ-Alias `Env`, das eine Abbildung von `String` nach `Int` bezeichnet.

```
import scala.language.implicitConversions

// ...
case class Id(x: String) extends Exp

implicit def num2exp(n: Int) : Exp = Num(n)
implicit def string2exp(s: String) : Exp = Id(s))

type Env = Map[String, Int]

def eval(e: Exp, env: Env) : Int = e match {
  case Num(n) => n
  case Add(l,r) => eval(l,env) + eval(r,env)
  case Mul(l,r) => eval(l,env) * eval(r,env)
  case Id(x) => env(x)
}

// example expressions
val exEnv = Map("x" -> 2, "y" -> 4)
val a = Add(Mul("x",5), Mul("y",7))
assert( eval(a,exEnv) == 38 )
val b = Mul(Mul("x", "x"), Add("x", "x"))
assert( eval(b,exEnv) == 16 )
```

Bei der rekursiven Auswertung der Unterausdrücke im `Add`- und `Mul`-Fall reichen wir die Umgebung `env` unverändert weiter. Um einen Identifier auszuwerten, schlagen wir ihn in der Umgebung nach und geben die mit ihm assoziierte Zahl aus.

Wir nutzen implizite Konvertierung, um Beispielausdrücke kompakter und lesbarer notieren zu können.

Abstraktion durch Visitor

Eine alternative Möglichkeit, den ersten Interpreter zu definieren, ist durch *Faltung* mit einem *internen Visitor*. Dabei handelt es sich um eine Instanz einer Klasse mit Typparameter `T`, die aus Funktionen mit den Typen `Int => T` und `(T,T) => T` besteht.

```
case class Visitor[T](num: Int => T, add: (T,T) => T)

def foldExp[T](v: Visitor[T], e: Exp) : T = e match {
```

```
  case Num(n) => v.num(n)
  case Add(l,r) => v.add(foldExp(v,l), foldExp(v,r))
}

val evalVisitor = new Visitor[Int](n => n, (l,r) => l+r)
val countVisitor = Visitor[Int](n => 1, (l,r) => l+r+1)
val printVisitor = Visitor[String](_.toString, "("+_+"+"+_+")")

assert( foldExp(evalVisitor, Add(2,4)) == 6 )
assert( foldExp(countVisitor, Add(2,4)) == 3 )
assert( foldExp(printVisitor, Add(2,4)) == "(2+4)")
```

Im Allgemeinen besteht ein interner Visitor für einen algebraischen Datentyp aus einer Funktion für jedes Konstrukt des Datentyps und einem Typparameter, der den Rückgabetyp der Faltung mit dem Visitor angibt. Der Typparameter wird an allen Stellen verwendet, an denen in der Definition des Datentyps der Datentyp selbst steht (im Beispiel die beiden Unterausdrücke von `Add`).

Ein wesentlicher Unterschied zur Pattern-Matching-Implementation ist, dass die Visitors selbst nicht rekursiv sind, sondern stattdessen das grundlegende Rekursionsmuster in einer Funktion abstrahiert wird (als Faltung, hier in `foldExp`). Dadurch können beliebige *kompositionale* Operationen auf der Datenstruktur (hier `Exp`) definiert werden, ohne dass weitere Funktionen notwendig sind.

Durch die Abstraktion des Rekursionsmusters wird *Kompositionalität* für alle Visitors erzwungen.

Kompositionalität heißt, dass sich die Bedeutung eines zusammengesetzten Ausdrucks aus der Bedeutung seiner Bestandteile ergibt. Bei einer rekursiven Struktur muss also die Bedeutungsfunktion strukturell rekursiv sein.

Eine Funktion ist **strukturell rekursiv**, wenn rekursive Aufrufe immer nur auf Unterausdrücken bzw. dem Inhalt der Datenfelder des aktuellen Ausdrucks stattfinden.

Auch in dieser Implementation ist es möglich, eine Core-Sprache und eine erweiterte Sprache zu definieren, in dem man etwa eine zweite Klasse definiert, die `Visitor` erweitert und um zusätzliche Funktionen (bspw. `sub: (T,T) => T`) ergänzt.

Wir können auch wieder Identifier durch eine Umgebung hinzufügen, der `eval`-Visitor muss dazu mithilfe von Currying verfasst werden. Für den Typparameter `T` des Visitors muss dann `Env => Int` gewählt werden, es wird erst ein Ausdruck und anschließend eine Umgebung eingelesen, bevor das Ergebnis ausgegeben wird. Dadurch lässt sich die Funktion trotz des zusätzlichen Parameters mit der Visitor-Klasse verfassen (hier verkürzt ohne `Mul`):

```
case class Visitor[T](num: Int => T, add: (T,T) => T, id: String => T)

def foldExp[T](v: Visitor[T], e: Exp) : T = e match {
  case Num(n) => v.num(n)
  case Add(l,r) => v.add(foldExp(v, l), foldExp(v, r))
  case Id(x) => v.id(x)
}

val evalVisitor = Visitor[Env=>Int](
  env => _ ,
  (a, b) => env => a(env) + b(env),
  x => env => env(x))
```

Identifier mit Bindings (WAE)

Bei der Implementation von Identifiern mit einer Environment müssen die Identifier außerhalb der Programme in der *Map* definiert werden und Identifier können nicht umdefiniert werden.

Besser wäre eine Implementation, bei der die Bindungen innerhalb von Programmen definiert und umdefiniert werden können. Dazu ist ein neues Sprachonstrukt, `With`, notwendig. Ein `With`-Ausdruck besteht aus einem Identifier, einem Ausdruck und einem Rumpf, in dem der Identifier an den Ausdruck gebunden ist.

```
case class With(x: String, xDef: Exp, body: Exp) extends Exp
```

`Add(5, With(x, 7, Add(x, 3)))` soll also bspw. zu `15` auswerten.

Die Bindung soll nur im Rumpf gelten, nicht außerhalb (*lexikalisches Scoping*).

Dazu soll die Definition von `x` (hier `7`) ausgewertet und für alle Vorkommen von `x` im Rumpf eingesetzt werden (*Substitution*). Hierfür definieren wir eine neue Funktion `subst` mit dem Typ `(Exp, String, Num) => Exp`. Es müssen zwei verschiedene Vorkommen von Identifiern unterschieden werden: *Bindende* Vorkommen (Definitionen in `With`-Ausdrücken) und *gebundene* Vorkommen (Verwendung an allen anderen Stellen). Tritt ein Identifier auf, ohne dass es "weiter außen" im Ausdruck ein bindendes Vorkommen gibt, so handelt es sich um ein *freies* Vorkommen.

```
With x = 7: // binding occurence
  x + y // x bound, y free
```

Der **Scope** (Sichtbarkeitsbereich) eines bindenden Vorkommens des Identifiers `x` ist die Region des Programmtextes in dem sich Vorkommen von `x` auf das bindende Vorkommen beziehen.

Beim Entwerfen der Sprache muss das erwünschte Scoping-Verhalten entschieden und implementiert werden. - Beispiel 1: Das gebundene Vorkommen von `x` soll sich auf die Definition in der ersten Zeile beziehen, die zweite Definition soll keine Wirkung "nach außen" haben.

```
With x = 5:
  x + (With x = 3: 10)
```

- Beispiel 2: Das erste Vorkommen soll hier (intuitiv, vgl. Scheme/Racket) den Wert 5 besitzen, dass zweite Vorkommen jedoch den Wert 3. Es soll also immer das nächste bzw. nächstinnerste bindende Vorkommen gelten.

```
With x = 5:
   x + (With x = 3: x)
```

Würde das Scope des ersten bindenden Vorkommens den gesamten Ausdruck umfassen, so wäre es nicht möglich, Identifier umzubinden, außerdem sind Programme so weniger verständlich und deren Auswertung schlechter nachvollziehbar, da Unterausdrücke je nach Kontext zu einem anderen Ergebnis auswerten würden.

Es ergibt sich die folgende Implementation für die Erweiterung um `With` und Substitution:

```
case class With(x: String, xDef: Exp, body: Exp) extends Exp

def subst(body: Exp, i: String, v: Num) : Exp = body match {
  case Num(_) => body
  case Id(x) => if (x == i) v else body
  case Add(l,r) => Add(subst(l,i,v), subst(r,i,v))
  case With(x,xDef,body) =>  // do not substitute in body if x is redefined
    With(x, subst(xDef,i,v), if (x == i) body else subst(body,i,v))
}

def eval(e: Exp) : Int = e match {
  case Num(n) => n
  case Id(x) => sys.error("Unbound identifier: " + x.name)
```

```
  case Add(l,r) => eval(l) + eval(r)
  case With(x,xDef,body) => eval(subst(body,x,Num(eval(xDef))))
}

val a = Add(5, With("x", 7, Add("x", 3)))
assert(eval(a) == 15)
val b = With("x", 1, With("x", 2, Add("x","x")))
assert(eval(b) == 4)
val c = With("x", 5, Add("x", With("x", 3, "x")))
assert((eval(c) == 8))
val d = With("x", 1, With("x", Add("x",1), "x"))
assert((eval(d) == 2))
```

Stößt die `eval`-Funktion auf einen Identifier, so ist dieser offensichtlich bei den bisherigen Substitutionen nicht ersetzt worden und ist frei. In diesem Fall wird also ein Fehler geworfen. Im `With`-Fall wird `subst` aufgerufen, um im Rumpf (`body`) die Vorkommen von `x` durch `xDef` zu ersetzen. Dabei wird `xDef` zuerst ausgewertet, was einen Wert vom Typ `Int` ergibt, und anschließend wieder mit `Num()` "verpackt", damit der erwartete Typ erfüllt ist und die Zahl als Unterausdruck eingefügt werden kann.

Bei der Substitution im `With`-Konstrukt darf die Substitution nur dann rekursiv im Rumpf angewendet werden, wenn der Identifier im `With` Konstrukt nicht gleich dem zu ersetzenden Identifier ist. In diesem Fall beziehen sich Vorkommen des Identifiers im Rumpf nämlich auf die Bindung im `With`-Konstrukt und nicht auf die Bindung, für die die Substitution durchgeführt wird.

Es muss aber immer im `xDef`-Ausdruck substituiert werden, denn auch hier können Identifier auftreten, die Definition aus dem `With`-Ausdruck soll aber nur im Rumpf und nicht in `xDef` gelten.

First-Order-Funktionen (F1-WAE)

Identifier ermöglichen Abstraktion bei mehrfach auftretenden, identischen Teilausdrücken (*"Magic Literals"*) – bspw. kann damit eine Konstante gebunden werden, die in einer Berechnung häufig vorkommt.

Unterscheiden sich die Teilausdrücke aber immer an einer oder an wenigen Stellen, so sind First-Order-Funktionen notwendig, um zu abstrahieren. Die Ausdrücke `5*3+1`, `5*2+1` und `5*7+1` lassen sich etwa mit `f(x) = 5*x+1` schreiben als `f(3)`, `f(2)` und `f(7)`. Weitere Beispiele dieser Abstraktion wären die Funktionen `square(n) = n*n` und `avg(x,y) = (x+y)/2`.

First-Order-Funktionen werden über einen Bezeichner aufgerufen, können aber nicht als Parameter übergeben werden und sind nicht Ausdrücke (Typ `Exp`).

Wir legen zwei Sprachkonstrukte für Funktionsaufrufe und Funktionsdefinitionen an. Aufrufe sind Expressions und bestehen aus dem Bezeichner der Funktion sowie einer Liste von Argumenten, Definitionen bestehen aus einer Liste von Parametern und einem Rumpf. In einer globalen Map werden Funktionsbezeichnern Funktionsdefinitionen zugewiesen, die Funktionen werden also außerhalb des Programms definiert (im Gegensatz zu Bindungen mit `With`).

```
case class Call(f: String, args: List[Exp]) extends Exp

case class FunDef(params: List[String], body: Exp)
type Funs = Map[String, FunDef]
```

Substitutionsbasierter Interpreter

Die bereits implementierten Bindungen mit `With` und die Funktionen verwenden getrennte *Namespaces*, es kann also der gleiche Bezeichner für eine Konstante und für eine Funktion verwendet werden, die Namensvergabe ist unabhängig voneinander (hier eine alternative Implementation mit geteiltem Namespace).

Wir erweitern `subst` um den `Call`-Fall, dabei wird die Substitution mit der `map`-Funktion auf alle Funktionsargumente angewandt.

```
def subst(body: Exp, i: String, v: Num) : Exp = body match {
  // ...
  case Call(f,args) => Call(f, args.map(subst(_,i,v)))
}
```

In `eval` überreichen wir zusätzlich immer die Map, in der Funktionen definiert sind. Der neue Match-Zweig ist hier deutlich komplizierter:

```
def eval(e: Exp, funs: Funs) : Int = e match {
  case Num(n) => n
  case Id(x) => sys.error("Unbound identifier "+x)
  case Add(l,r) => eval(l,funs)+eval(r,funs)
  case With(x,xDef,b) => eval(subst(b,x,Num(eval(xDef,funs))), funs)
  case Call(f,args) => {
    val fDef = funs(f)
    val vArgs = args.map(eval(_,funs))
    if (fDef.params.size != vArgs.size)
      sys.error("Incorrect number of args in call to "+f)
    val substBody = fDef.params.zip(vArgs).foldLeft(fDef.body)(
      (b,pv) => subst(b, pv._1, Num(pv._2))
    )
    eval(substBody,funs)
  }
}
```

`f` ist ein Bezeichner vom Typ `String`, `args` ist die Liste der Argumente des Aufrufs vom Typ `List[Exp]`.

Zuerst wird die Definition von `f` in `funs` nachgeschlagen und das Ergebnis an `fDef` gebunden. Mit `map` werden dann alle Argumente in der Liste vollständig ausgewertet. `vArgs` ist die Liste der ausgewerteten Argumente mit Einträgen vom Typ `Int`. Als nächstes wird geprüft, dass die Argumentliste `vArgs` und die Parameterliste `fDef.params` die selbe Länge besitzen, ist dies nicht der Fall, so wird ein Fehler geworfen.

Nun wird für jeden Parameter in der Parameterliste die Substitution im Rumpf `fDef.body` mit dem entsprechenden Argument aus `vArgs` ausgeführt. Dies ist durch `zip` und `foldLeft` implementiert. `fDef.params.zip(vArgs)` erzeugt eine Liste vom Typ `List[(String,Int)]`, in der jeder Parameter mit dem korrespondierenden Argument in einem Tupel vorliegt. `foldLeft` erhält `body` als Startwert und wendet dann (vom Ende der Liste beginnend) nacheinander für jedes Tupel die entsprechende Substitution an. Es werden also Aufrufe von `subst` geschachtelt, wobei die innereste Substitution auf dem ursprünglichen Rumpf `body` ausgeführt wird.

Zuletzt wird `eval` rekursiv auf dem Rumpf, in dem alle Substitutionen durchgeführt wurden, aufgerufen.

Es ist nun möglich, nicht-terminierende Programme zu verfassen. Wird in der Definition einer Funktion die Funktion selbst aufgerufen, so entsteht eine Endlosschleife.

```
val fm = Map("square" -> FunDef(List("x"), Mul("x","x")),
             "succ" -> FunDef(List("x"), Add("x",1)),
             "myAdd" -> FunDef(List("x","y"), Add("x","y")),
             "forever" -> FunDef(List("x"), Call("forever", List("x"))))

val a = Call("square", List(Add(1,3)))
assert(eval(fm,a) == 16)
val b = Mul(2, Call("succ", List(Num(20))))
assert(eval(fm,b) == 42)
val c = Call("myAdd", List(Num(40), Num(2)))
```

```
assert(eval(fm,c) == 42)

val forever = Call("forever", List(Num(0)))
// eval(fm,forever) does not terminate
```

Umgebungsbasierter Interpreter

Unsere bisherige Implementierung von Substitution würde im Ausdruck

```
With("x", 1, With("y", 2, With("z", 3, Add("x", Add("y", "z")))))
```

folgende Schritte durchlaufen:

```
With("y", 2, With("z", 3, Add(1, Add("y", "z"))))
// ~~>
With("z", 3, Add(1, Add(2, "z")))
// ~~>
Add(1, Add(2, 3))
```

Dabei wird der Ausdruck `Add("x", Add("y", "z"))` insgesamt drei Mal traversiert, um für jedes `With` die Substitution durchzuführen. Die Komplexität bei Ausdrücken der Länge n ist $\mathcal{O}(n^2)$. Wir suchen deshalb eine effizientere Art, um Substitution umzusetzen.

Statt beim Auftreten eines `With`-Ausdrucks direkt zu substituieren, wollen wir uns in einer zusätzlichen Datenstruktur merken, welche Substitutionen wir im weiteren Ausdruck vornehmen müssen, so dass der zusätzliche Durchlauf wegfällt.

Hierzu verwenden wir wieder (wie in AEId) eine *Umgebung*, diese wird aber nicht getrennt und global definiert, sondern bei der Evaluation stetig angepasst. Tritt etwa ein `With`-Ausdruck auf, so wird der entsprechende Identifier mit dem Auswertungsergebnis der zugewiesenen Expression in die Map eingetragen (die zu Beginn der Auswertung leer ist).

```
type Env = Map[String, Int]

def evalWithEnv(funs: FunDef, env: Env, e: Exp) : Int = e match {
  case Num(n) => n
  case Id(x) => env(x)
  case Add(l,r) => evalWithEnv(funs,env,l) + evalWithEnv(funs,env,r)
  case With(x,xDef,body) =>
    evalWithEnv(funs, env+(x -> evalWithEnv(funs,env,xDef)), body)
}
```

Da nun die notwendigen Substitutionen nicht direkt ausgeführt, sondern in der Umgebung hinterlegt und "aufgeschoben" werden, können wir auf nicht-substituierte Identifier stoßen. Diese schlagen wir dann in der Umgebung `env` nach, um sie durch den Wert zu ersetzen, an den sie gebunden sind.

Das vorherige Beispiel wird nun folgendermaßen ausgewertet:

```
With("x", 1, With("y", 2, With("z", 3, Add("x", Add("y", "z"))))), Map()
// ~~>
With("y", 2, With("z", 3, Add("x", Add("y", "z")))), Map("x" -> 1)
// ~~>
With("z", 3, Add("x", Add("y", "z"))), Map("x" -> 1, "y" -> 2)
// ~~>
Add("x", Add("y", "z")), Map("x" -> 1, "y" -> 2, "z" -> 3)
```

Die Komplexität ist nun (unter der Annahme, dass die Map-Operationen in konstanter Komplexität haben) linear in Abhängigkeit von der Länge des Ausdrucks.

Das Scoping ist auch in dieser Implementation lexikalisch, da die Umgebung rekursiv weitergereicht wird und nicht global ist. Somit gilt eine Bindung nur in Unterausdrücken des bindenden Ausdruckes und nicht an anderen Stellen im Programm. Die +-Operation auf Maps fügt nicht nur Bindungen für neue Elemente ein, sondern ersetzt auch den Abbildungswert bei bereits enthaltenen Elementen:

```
var m = Map("a" -> 1)
m = m+("b" -> 2)
m = m+("a" -> 3)
// ~~>
m: Map[String,Int] = Map("a" -> 3, "b" -> 2)
```

Nun fehlt noch der `Call`-Fall. Hier bleiben die ersten drei Zeilen nahezu identisch, aber anstelle der Faltung zur Substitution im Rumpf erweitern wir einfach die leere Umgebung um die Parameternamen, gebunden an die ausgewerteten Argumente:

```
def evalWithEnv(funs: FunDef, env: Env, e: Exp) : Int = e match {
// ...
  case Call(f,args) =>
    val fDef = funs(f)
    val vArgs = args.map(evalWithEnv(funs,env,_))
    if (fDef.args.size != vArgs.size)
      sys.error("Incorrect number of params in call to " + f)
    evalWithEnv(funs, Map()++fDef.args.zip(vArgs), fDef.body)
}
```

Dabei können mit dem Operator `++` die Tupel in der Liste der Map hinzugefügt werden.

Wir erweitern die leere Umgebung `Map()` anstelle der bisherigen Umgebung `env`, da in unserer vorherigen Implementation auch nur für die Funktionsparameter im Funktionsrumpf substituiert wurde (und nicht für sonstige, aktuell geltende Bindungen). Die `subst`-Funktion verändert nämlich die Funktionsdefinitionen in keiner Weise und hat nicht einmal Zugriff auf diese.

Äquivalenz des substitutions- und umgebungsbasierten Interpreters:

Für alle `funs: Funs, e: Exp` gilt: `evalWithSubst(funs,e) = evalWithEnv(funs,Map(),e)`.

Nun gäbe es aber auch die Möglichkeit, die bisherige Umgebung `env` zu erweitern und damit den Funktionsrumpf auszuwerten. In diesem Fall würden wir lokale Bindungen, die an der Stelle des `Call`-Ausdrucks gelten, in den Funktionsrumpf weitergeben.

Die Variante mit einer neuen, leeren Umgebung wird *lexikalisches Scoping* genannt, die Variante bei der `env` erweitert wird heißt *dynamisches Scoping*.

Lexikalisches und dynamisches Scoping

Lexikalisches (oder statisches) Scoping bedeutet, dass der Scope eines bindenden Vorkommens syntaktisch beschränkt ist, bspw. auf einen Funktionsrumpf. In unserer Sprache wird für ein Vorkommen eines Identifiers der Wert durch das erste bindende Vorkommen auf dem Weg vom Identifier zur Wurzel des abstrakten Syntaxbaums (AST) bestimmt.

Dynamisches Scoping bedeutet, dass für ein Vorkommen eines Identifiers der Wert durch das zuletzt ausgewertete bindende Vorkommen bestimmt wird. Eine Bindung gilt dadurch während der gesamten weiteren Programmausführung und ist nicht auf einen bestimmten Bereich im Programms beschränkt.

Bei lexikalischem Scoping ist also der Ort für die Bedeutung entscheidend, bei dynamischem Scoping der Programmzustand.

Das folgende Beispiel verursacht bei lexikalischem Scoping einen Fehler, liefert aber bei dynamischem Scoping das Ergebnis `3`:

```
val exFunMap = Map("f" -> FunDef(List("x"), Add("x","y")))
val exExpr = With("y", 1, Call("f", List(2)))

evalWithEnv(exFunMap, Map(), exExpr)
```

Der Identifier `y` wird an den Wert `1` gebunden, bevor die Funktion `f` aufgerufen wird, in deren Rumpf `y` auftritt. `y` hat an diesem *Ort* im Programm keine Bedeutung, es kann aber ein *Programmzustand* vorliegen, in dem eine Bindung für `y` existiert.

Bei lexikalischem Scoping müssen Werte immer "weitergereicht" werden, während bei dynamischen Scoping alle Bindungen zum Zeitpunkt eines Funktionsaufrufs auch im Funktionsrumpf gelten. Diese automatische Weitergabe kann in manchen Fällen ein explizites Überreichen ersparen, führt aber in den meisten Fällen eher zu unerwarteten und unerwünschten Nebenwirkungen.

Ein Beispiel für eine Verwendung von dynamischem Scoping wäre *Exception Handling* in Java. Wird in einem *try-catch*-Block eine Funktion `f` aufgerufen, die eine bestimmte Exception wirft, so wird beim Werfen dieser Exception über eine Art von dynamischem Scoping ermittelt, welcher ExceptionHandler zuständig ist (in dem die Ausführungshistorie durchsucht wird).

Higher-Order-Funktionen (FAE)

Funktionen erster Ordnung erlauben die Abstraktion über sich wiederholende Muster, die an bestimmten Ausdruckspositionen variieren (z.B. eine `square`- oder `avg`-Funktion). Liegt aber ein Muster vor, bei dem eine Funktion variiert (z.B. bei der Komposition zweier Funktionen), so ist keine Abstraktion möglich.

Hierfür sind Higher-Order-Funktionen notwendig, es braucht eine Möglichkeit, Funktionen als Parameter zu übergeben, als Ergebnis zurückzugeben und als Werte zu behandeln. Wir müssen also unsere Implementation anpassen, so dass Funktionen nicht als ein vom Programm getrenntes Konstrukt, sondern als Expressions vorliegen.

Wir entfernen also das Sprachkonstrukt `Call` und ergänzen stattdessen die zwei folgenden Konstrukte:

```
case class Fun(param: String, body: Exp) extends Exp
case class App(funExpr: Exp, argExpr: Exp) extends Exp
```

Nun können wir Funktionen direkt im Programm definieren und binden, wodurch wir keine getrennte `Funs`-Map mehr benötigen. Funktion sind eine Form von Werten, solche "namenslosen" Funktionen werden typischerweise *anonyme Funktionen* genannt, im Kontext funktionaler Sprachen auch *Lambda-Ausdrücke*.

`With` ist jetzt sogar nur noch syntaktischer Zucker, wir können bspw. `With("x", 5, Add("x",7))` ausdrücken mit `App(Fun("x", Add("x",7)), 5)`.

```
def wth(x: String, xdef: Exp, body: Exp) : Exp = App(Fun(x, body), xdef)
```

Ein `Fun`-Ausdruck hat nur einen Parameter und ein `App`-Ausdruck nur ein Argument (im Gegensatz zu unserer Implementation von First-Order-Funktionen), wir können jedoch Funktionen mit mehreren Parametern durch Currying darstellen: $f(x, y) = x + y$ entspricht $f(x)(y) = x + y$ bzw. in der Notation des Lambda-Kalküls $\lambda x.\lambda y.x + y$.

Accidental Captures

Zuerst implementieren wir den Interpreter wieder durch Substitution:

```
def subst(e: Exp, i: String, v: Exp) : Exp = e match {
  case Num(_) => e
  case Id(x) => if (x == i) v else e
  case Add(l,r) => Add(subst(l,i,v), subst(r,i,v))
  case Fun(p,b) =>
    if (param == i) e else Fun(p, subst(b,i,v))
```

```
  case App(f,a) => App(subst(f,i,v), subst(a,i,v))
}
```

Hierbei entsteht ein neues Problem: Der zu substituierende Ausdruck `v` muss nun den Typ `Exp` besitzen, damit Identifier auch durch Funktionen (und nicht nur `Num`-Ausdrücke) substituiert werden können. Dadurch kann es aber in manchen Fällen dazu kommen, dass Identifier unbeabsichtigt gebunden werden:

```
val ac = subst(Fun("x", Add("x","y")), "y", Add("x",5))
```

In diesem Beispiel ist das `x` in `Add(x, 5)` nach der Substitution an den Parameter `x` der Funktion gebunden, obwohl dies vorher nicht der Fall war. Die dabei entstehende Bindung ist unerwartet, dieses unerwünschte "Einfangen" eines Identifiers wird als *Accidental Capture* bezeichnet und allgemein als Verletzung von lexikalischem Scoping angesehen.

Capture-Avoiding Substitution

Zwei Funktionen sind **alpha-äquivalent**, wenn sie bis auf den Namen des Parameters (oder der Parameter) identisch sind. `Fun("x", Add("x",1))` und `Fun("y", Add("y",1))` sind bspw. *alpha-äquivalent*.

Wir nutzen Alpha-Äquivalenz, um Namenskonflikte und damit Accidental Captures zu verhindern. Dabei wählen wir für den Parameter einer Funktion einen neuen Namen, der weder im zu substituierenden Ausdruck noch im aktuellen Ausdruck ungebunden auftritt. Wir brauchen also einen "Generator", um bisher ungenutzte Namen zu erzeugen, die wir dann zur Umbenennung verwenden können.

```
def freshName(names: Set[String], default: String) : String = {
  var last : Int = 0
  var freshName = default
  while (names contains freshName) {
    freshName = default + last
    last += 1
  }
  freshName
}
```

Die Funktion gibt einen Namen zurück, der nicht in der Menge `names` enthalten ist. Ist `default` nicht in der Menge enthalten, so wird `default` ausgegeben, ansonsten wird eine Zahl an die Eingabe `default` angehängt und schrittweise inkrementiert, bis der entstehende String nicht Element der Menge ist.

Wir benötigen außerdem eine Funktion, die die Menge aller freien Variablen in einem Ausdruck ausgibt, dazu verwenden wir den Datentyp `Set`:

```
def freeVars(e: Exp) : Set[String] = e match {
  case Num(_) => Set.empty
  case Id(x) => Set(x)
  case Add(l,r) => freeVars(l) ++ freeVars(r)
  case App(f,a) => freeVars(f) ++ freeVars(a)
  case Fun(x,body) => freeVars(body) - x
}

assert(freeVars(Fun("x"), Add("x","y")) == Set("y"))
```

Mithilfe dieser Funktionen können wir nun beim Substituieren Accidental Captures verhindern, man spricht hierbei von *Capture-Avoiding Substitution*:

```
def subst(e: Exp, i: String, v: Exp) : Exp = e match {
  case Num(_) => e
  case Add(l,r) => Add(subst(l,i,v), subst(r,i,v))
  case Id(x) => if (x == i) v else e
  case Fun(p,b) =>
```

```
    if (p == i) e else {
      val fvs = freeVars(e) ++ freeVars(v) + i
      val nn = freshName(fvs,p)
      Fun(nn, subst(subst(b,p,Id(nn)), i, v))
    }
  case App(f: Exp, a: Exp) => App(subst(f,i,v), subst(a,i,v))
}
```

Im `Fun`-Fall prüfen wir zuerst, ob der Parameter und der zu ersetzende Identifier übereinstimmen. Ist dies der Fall, so lassen wir den `Fun`-Ausdruck unverändert, da der Rumpf nicht im Scope des bindenden Vorkommens liegt, für das substituiert wird.

Ansonsten bestimmen wir mit `freeVars` die Menge der freien Variablen im aktuellen Ausdruck `e` sowie im einzusetzenden Ausdruck `v` und vereinigen diese zusammen mit `i` (damit nicht `i` als neuer Name gewählt und fälschlicherweise substituiert wird, falls `i` nicht in `e` frei vorkommt). Ausgehend von dieser Menge erzeugen wir mit `freshName` einen neuen Bezeichner, mit dem wir dann den Parameternamen und alle Vorkommen des Parameternamens im Rumpf ersetzen, bevor wir die Substitution im Rumpf rekursiv fortsetzen. So ist garantiert, dass keine freien Variablen durch die Funktion "eingefangen" werden.

Substitutionsbasierter Interpreter

Da Funktionen nun Werte bzw. Ausdrücke sind, muss der Interpreter auch Funktionen als Ergebnis einer Auswertung ausgeben können. Der Rückgabewert von `eval` kann also nicht mehr `Int` sein, stattdessen müssen wir `Exp` wählen.

Durch diesen Rückgabetyp gibt es aber auch eine neue Klasse von Fehlern, die auftreten können: Es kann passieren, dass eine Zahl erwartet wird, aber eine Funktion vorliegt, etwa wenn der linke Teil eines `Add`-Ausdrucks zu einer Funktion auswertet. Auch der umgekehrte Fall kann eintreten: In einem `App`-Ausdruck wertet der linke Teil zu einer Zahl anstelle einer Funktion aus.

```
def eval(e: Exp) : Exp = e match {
  case Id(x) => sys.error("Unbound identifier: " + x)
  case Add(l,r) => (eval(l),eval(r)) match {
    case (Num(a),Num(b)) => Num(a+b)
    case _ => sys.error("Can only add numbers")
  }
  case App(f,a) => eval(f) match {
    case Fun(param,body) => eval(subst(body,param,eval(a))) // call-by-value
    // case Fun(param,body) => eval(subst(body,param,a))    // call-by-name
    case _ => sys.error("Can only apply functions")
  }
  case _ => e
}
```

Aus diesem Grund müssen wir im `Add`- und `App`-Fall erst beide Unterausdrücke auswerten und dann via Pattern-Matching prüfen, ob jeweils der korrekte Typ vorliegt. Ist dies nicht der Fall, so werfen wir einen Fehler. `Num`- und `Fun`-Ausdrücke werten zu sich selbst aus, die Ausgabe des Interpreters ist immer vom Typ `Num` oder `Fun`.

Wir könnten den Rückgabetyp also mit `Either[Num,Fun]` präzisieren (mögliche Implementation).

Umgebungsbasierter Interpreter

Der Typ `Map[String,Int]` für die Umgebung ist nicht mehr ausreichend, da auch `Fun`-Ausdrücke gebunden werden müssen. Wir wählen also stattdessen:

```
type Env = Map[String,Exp]
```

Der Interpreter ergibt sich größtenteils aus der Implementation für F1-WAE, kombiniert mit dem substitutionsbasiertem Interpreter für FAE:

```
def eval(e: Exp, env: Env) : Exp = e match {
  case Add(l,r) => (eval(l,env),eval(r,env)) match {
    case (Num(a),Num(b)) => Num(a+b)
    case _ => sys.error("Can only add numbers")
  }
  case Id(x) => env(x)
  case App(f,a) => eval(f,env) match {
    case Fun(p,b) =>
      eval(b, Map(p -> eval(a,env)))
    case _ => sys.error("Can only apply functions")
  }
  case _ => e
}
```

Wie in F1-WAE erweitern wir bei der Rekursion in einen Funktionsrumpf die Umgebungs nicht, da sonst dynamisches Scoping vorliegen würde. In diesem Fall würde der Ausdruck

`With` $f = \lambda x.x + y$: `With` $y = 4$: $(f\ \ 1)$

keinen Fehler verursachen, obwohl zum Zeitpunkt der Bindung von f der Bezeichner y frei ist.

Stattdessen reichen wir also eine neue Umgebung weiter, die nur den Parameter und das Argument der Funktion enthält.

Dieser Interpreter ist jedoch nicht äquivalent zum substitutionsbasierten, denn der Ausdruck

```
val curry = App( Fun("x", App( Fun("y", Add("x","y")),2)), 3)
```

wird von `evalWithSubst` zu `Num(5)` ausgewertet, `evalWithEnv` wirft aber bei der Auswertung den Fehler `key not found: x`:

$(\lambda x.(\lambda y.x+y\ \ 2)\ \ 3)$, `Map()` $\rightsquigarrow (\lambda y.x+y\ \ 2)$, `Map(x -> 3)` $\rightsquigarrow x+y$, `Map(y -> 2)` $\rightsquigarrow$ `key not found: x`

Um dieses Problem zu umgehen, können wir nicht einfach die Umgebung im `App`-Fall erweitern, weil das (wie bereits gezeigt) zu dynamischem Scoping führen würde. Stattdessen müssen wir uns bei der Auswertung einer Funktion sowohl die Funktion selbst, als auch die Umgebung zum Zeitpunkt der Instanziierung merken.

So ein Paar aus Funktion und Umgebung wird *Closure* genannt.

Closures

Wir definieren einen neuen Typ `Value` neben `Exp`, so dass wir Ausdrücke und die Ergebnisse deren Auswertung wieder unterscheiden können:

```
sealed abstract class Value
type Env = Map[String,Value]
case class NumV(n: Int) extends Value
case class ClosureV(f: Fun, env: Env) extends Value
```

`Fun`-Ausdrücke liegen nun nach ihrer Auswertung als Closures vor, der neue Interpreter hat den Rückgabetyp `Value` und gibt Zahlen (`NumV`) und Closures (`ClosureV`) anstelle von Zahlen (`Num`) und Funktionen (`Fun`) aus.

```
def eval(e: Exp, env: Env) : Value = e match {
  case Num(n) => NumV(n)
  case Id(x) => env(x)
  case Add(l,r) => (eval(l,env),eval(r,env)) match {
    case (NumV(a),NumV(b)) => NumV(a+b)
    case _ => sys.error("Can only add numbers")
```

```
  }
  case f@Fun(_,_) => ClosureV(f,env)
  case App(f,a) => eval(f,env) match {
    case ClosureV(Fun(p,b),cEnv) =>
      eval(b, cEnv+(p -> eval(a,env)))
    case _ => sys.error("Can only apply functions")
  }
}
```

Bei der Auswertung eines `Fun`-Ausdrucks wird nun ein Closure aus dem Ausdruck und der aktuellen Umgebung erzeugt. Im `App`-Zweig verwenden wir nach der Auswertung der Funktion die Umgebung aus dem entstehenden Closure – erweitert um die Bindung des Parameters an das ausgewertete Argument – um den Rumpf auszuwerten.

Durch das "Aufschieben" der notwendigen Substitutionen in der Umgebung ist es also im Fall von Funktionen notwendig geworden, die aufgeschobenen Substitutionen für die spätere Auswertung zu hinterlegen.

Für alle `e: Exp` gilt: - `evalWithSubst(e) == Num(n)` $\iff$ `evalWithEnv(e,Map()) == NumV(n)`

- Ist `evalWithSubst(e) == Fun(p,b)` und `evalWithEnv(e) == ClosureV(f,env)`, dann entspricht `Fun(p,b)` dem Ausdruck `f`, in dem für alle Bindungen in `env` Substitution durchgeführt wurde.

Closures sind ein fundamental wichtiges Konzept und tauchen in der Implementation fast aller Programmiersprachen auf.

Ein **Closure** ist ein Paar, bestehend aus einer Funktionsdefinition und der Umgebung, die bei der Auswertung der Funktionsdefinition aktiv bzw. gültig war.

Im substitiutionsbasiertem Interpreter wird beim Auswerten der Funktionsdefinition sofort im Rumpf substituiert. Beim umgebungsbasiertem Interpreter muss hingegen die Umgebung zum Zeitpunkt der Auswertung gespeichert werden, so dass der Interpreter beim Erreichen der Identifier die richtigen Werte einsetzen kann.

Mächtigkeit von FAE

Wir können in FAE nicht-terminierende Ausdrücke verfassen:

```
val omega = App( Fun("x", App("x","x")), Fun("x", App("x","x")) )
```

In `eval` betreten wir den `App`-Fall, dort wird substituiert, wodurch wieder `omega` entsteht.

```
assert(subst(App("x","x"), "x", Fun("x", App("x","x"))) ==
App(Fun("x", App("x","x")), Fun("x", App("x","x"))))
```

`omega` wird im *Lambda-Kalkül* notiert als $(\lambda x.(x\ x)\ \ \lambda x.(x\ x))$, der gesamte vordere Ausdruck wird auf den hinteren Ausdruck angewendet, der hintere Ausdruck wird in den Rumpf des vorderen Ausdrucks für x eingesetzt, wodurch wieder der ursprüngliche Ausdruck entsteht.

FAE ist Turing-mächtig, es können also alle Turing-berechenbaren Funktionen repräsentiert werden. Die Sprache entspricht nämlich dem von Alonzo Church eingeführten Turing-mächtigen Lambda-Kalkül, erweitert um Zahlen und Addition.

Lambda-Kalkül

Entfernen wir aus FAE den `Num`- und den `Add`-Fall, so erhalten wir eine Sprache, die dem *Lambda-Kalkül* entspricht:

```
sealed abstract class Exp
case class Id(name: String) extends Exp
case class Fun(param: String, body: Exp) extends Exp
case class App(fun: Exp, arg: Exp) extends Exp
```

Der Lambda-Kalkül ist Turing-vollständig, es können darin beliebige Berechnungen ausgedrückt werden. In seiner reinen Form ist es nicht unbedingt eine praktische oder sinnvolle Sprache, dennoch ist der Lambda-Kalkül von einem theoretischen Standpunkt relevant und betrachtenswert.

In dieser Sprache ist jeder Wert eine Funktion (bzw. ein Closure), wodurch keine Typfehler auftreten können.

```
abstract class Value
type Env = Map[String, Value]
case class ClosureV(f: Fun, env: Env) extends Value
```

Um mit dieser minimalistischen Sprache sinnvoll arbeiten zu können, sind Kodierungen für verschiedene Datentypen notwendig, dazu kann man bspw. die sogenannten *Church Encodings* verwenden.

Church-Kodierungen

Booleans werden als ihre "eigene If-Then-Else-Funktion"1 definiert. Darauf basierend lassen sich diverse Bool'sche Operationen definieren:

```
val t = Fun("t", Fun("f","t")) // true
val f = Fun("t", Fun("f","f")) // false

val ifTE = Fun("c", Fun("t", Fun("e", App(App("c","t"),"e"))))
val not = Fun("a", App(App("a", f), t)) // if a then False else True
val and = Fun("a", Fun("b", App(App("a", "b"), f))) // if a then b else False
val or = Fun("a", Fun("b", App(App("a", t), "b")))  // if a then True else b
```

Natürliche Zahlen können wir als Peano-Zahlen durch eine Nullfunktion und eine Nachfolgerfunktion kodieren. Dabei wird die Zahl n dargestellt durch die n-fache Anwendung einer Funktion s auf einen Startwert z.

```
val zero = Fun("s", Fun("z","z"))
val succ = Fun("n", Fun("s", Fun("z", App("s", App(App("n", "s"), "z")))))
val one = App(succ,zero) // = Fun("s", Fun("z", App("s","z")))
val two = App(succ,one) // = Fun("s", Fun("z", App("s", App("s","z"))))
val three = App(succ,two)

val add =
  Fun("a", Fun("b", Fun("s", Fun("z",
    App(App("a","s"), App(App("b","s"),"z"))))))
val mul =
  Fun("a", Fun("b", Fun("s", Fun("z",
    App(App("a", App("b","s")), "z")))))
```

In der `succ`-Funktion wird der Ausdruck erst "ausgepackt", indem er auf das s und dann auf das z angewendet wird, dann wird der zusätzliche Aufruf von s hinzugefügt und zuletzt wird der Ausdruck wieder zwei Mal in eine Funktion geschachtelt, um s und z wieder zu parametrisieren.

Bei der Addition wird der Startwert für a durch b ersetzt, wodurch die `succ`-Operation a-Mal auf b durchgeführt wird, bei der Multiplikation wird a-Mal b auf den Startwert addiert.

Durch ein Spachkonstrukt `printDot()`, der bei der Auswertung die Identitätsfunktion ausgibt und einen Punkt druckt, lassen sich durch den folgenden Ausdruck die unären Zahlenkodierungen visualisieren:

```
val printNum = Fun("n", App(App("n", Fun("x",printDot())), f))
```

Wir können auch mit einer Funktion prüfen, ob eine Zahl 0 ist. Da `zero` als `Fun("s", Fun("z","z"))` kodiert ist, liefert der folgende Ausdruck für 0 `True`, für Zahlen größer 0 die n-fache Anwendung von $\lambda x.$`False` auf `t`, also `False`.

```
val isZero = Fun("n", App(App("a", Fun("x",f)),t))
```

Es können auch eine Vorgängerfunktion `pred` und negative Zahlen mithilfe des sogenannten *Pairing Trick* kodiert werden, was jedoch deutlich aufwändiger ist.

Auch **Listen** können im Lambda-Kalkül kodiert werden. Die Grundidee ist es, zur Repräsentation die leere Liste e und wiederholte Anwendungen von c (`cons`) zu nutzen. Die Liste $x_1, x_2, ..., x_n$ wird also durch $\lambda c.\lambda e.c(x_1, c(...c(x_{n-1}, c(x_n, e))...))$ kodiert.

```
val empty = Fun("c", Fun("e","e"))
val cons =
  Fun("h", Fun("r", Fun("c", Fun("e",
    App(App("c","h"), App(App("r","c"), "e"))))))
```

Die leere Liste `empty` besitzt die gleiche Kodierung wie `zero` und `f`. Bei der `cons`-Operation wird die Restliste `r` durch Applikation auf `c` und `e` "entpackt", durch Anwendung von `c` auf das Element `h` und die "entpackte" Restliste wird das neue Kopfelement vorne an die Restliste angefügt.

```
val list123 = App(App(cons,one), App(App(cons,two), App(App(cons,three), empty)))
val listSum = Fun("l", App(App("l",add), zero))
```

Durch Applikationen von `cons` mit jeweils einem Kopfelement und einer Restliste sowie der leeren Liste `empty` lassen sich Listen kodieren, wie es bei `list123` zu sehen ist. Die Applikation einer Liste mit einem `c` und einem `e` entspricht der Listenfaltung, wie sie etwa in Scala oder Racket bzw. Scheme möglich ist:

```
List(1,2,3).fold(0)(_+_)
```

```
(foldr + 0 (list 1 2 3))
```

Listen werden also sozusagen durch ihre Fold-Funktion (also durch ihre Faltung mit den Argumenten `c` und `e`) repräsentiert.

Es können auch Listen von Listen kodiert werden, womit es eine Repräsentation von Bäumen im Lambda-Kalkül gibt. Wie wir an unserer eigenen Implementierung sehen können, lassen sich Programme im Lambda-Kalkül sehr gut durch Baumstrukturen darstellen. Dadurch lassen sich auch Lambda-Kalkül-Ausdrücke im Lambda-Kalkül selbst geschickt repräsentieren und es ist möglich, ein Programm im Lambda-Kalkül zu schreiben, das Ausdrücke im Lambda-Kalkül auswertet (vgl. universelle Turing-Maschine).

Rekursion

Es ist auch möglich, Rekursion im Lambda-Kalkül zu implementieren. Aus dem Programm `omega = (x => x x) (x => x x)` lässt sich das Programm `Y f = (x => f (x x)) (x => f (x x))` konstruieren, mit dem Schleifen kodiert werden können. Das Programm `Y` ist ein sogenannter *Fixpunkt-Kombinator*.

Ein **Fixpunkt-Kombinator** ist ein Programm, mit dem der Fixpunkt von Funktionen gebildet werden kann.

Bei der Auswertung von `Y f` entsteht ein Ausdruck der Form `(f ... (f (f (f Y)))...)`.

Lazy Evaluation/Call-By-Name (LCFAE)

Im substitutionsbasierten Interpreter für FAE konnten wir im `App`-Fall zwischen zwei möglichen Implementation wählen: Wir können das Argument vor der Substitution im Rumpf auswerten, oder substituieren, ohne das Argument vorher auszuwerten.

Die erste dieser *Auswertungsstrategien* wird *Call-By-Value* genannt, die zweite *Call-By-Name*.

Im substitutionsbasierten Interpreter gilt für alle `e: Exp`:

Ist `evalCBV(e) == e1` und `evalCBN(e) == e2`, dann sind `e1` und `e2` äquivalent (falls Zahlen – identisch, falls Funktionen – gleichbedeutend).

`evalCBV(App(Fun("x",0), omega))` terminiert nicht, `evalCBN(App(Fun("x",0), omega))` liefert hingegen das Ergebnis `Num(0)`. Der Call-By-Name-Interpreter terminiert auf strikt mehr Programmen als der Call-By-Value-Interpreter.

Mit "gleichbedeutend" ist gemeint, dass sich die Funktionen gleich verhalten, aber nicht zwingend identisch sind. Für den Ausdruck $(\lambda x.(\lambda y.x + y) \quad 1 + 1)$ würde Call-By-Value-Auswertung das Ergebnis $\lambda y.2 + y$ liefern, während Call-By-Name-Auswertung das Ergebnis $\lambda y.(1 + 1) + y$ liefern würde. Unterscheiden sich die ausgegebenen Funktionen, so ist die Funktion im Ergebnis von `evalCBV` stets "weiter ausgewertet".

Nutzen von Lazy Evaluation

Die Auswertungsstrategie eines Interpreters bzw. einer Programmiersprache ist nicht nur eine Frage der Effizienz, sondern hat auch Auswirkungen darauf, welche Programmstrukturen möglich sind. Unendliche Datencontainer wie Streams sind bspw. nur durch *Lazy Evaluation* überhaupt implementierbar.

In der Sprache Haskell kann man etwa eine rekursive Funktion ohne Abbruchkriterium schreiben, die die Quadratwurzel einer Zahl annähert.

Da Haskell *lazy* ist, kann eine Funktion ohne Abbruchbedingung definiert und mit verschiedenen Abbruchbedingungen aufgerufen werden, eine Programmstruktur die durch die Auswertungsstrategie der Sprache ermöglicht wird.

Der `evalCBN`-Interpreter erlaubt uns die Kodierung unendlicher Listen in LCFAE durch Church-Encodings (Zusatzmaterial in `08-lcfae.scala`).

Thunks

Im substitutionsbasierten Interpreter muss nur ein Funktionsaufruf entfernt werden, um die Auswertungsstrategie zu wechseln. Im umgebungsbasierten Interpreter müssen wir dagegen einige Änderungen vornehmen.

Wir müssen bei Funktionsapplikation den Parameter in der Umgebung an das Argument ohne vorherige Auswertung binden. Dabei stoßen wir auf das gleiche Problem, das uns bei Funktionen begegnet ist: Im Argument-Ausdruck, den wir an den Parameternamen binden, müssen noch Bezeichner substituiert werden, wozu die aktuelle Umgebung benötigt wird. Auf diese kann aber zum Zeitpunkt, zu dem das Argument ausgewertet wird, nicht mehr zugegriffen werden.

Analog zu den Closures für Funktionen brauchen wir also eine Datenstruktur, in der wir sowohl den Ausdruck, als auch die aktuelle Umgebung ablegen. Solch ein Paar aus `Exp` und `Env` nennen wir `Thunk`.

Die intutive Definition

```
type Thunk = (Exp, Env)
type Env = Map[String, Thunk]
```

ist in Scala durch die gegenseitige Bezüglichkeit nicht möglich. Stattdessen definieren wir den Interpreter als `trait`, wobei der Typ `Thunk` und die Funktionen `delay` und `force` abstrakt sind, so dass wir verschiedene Implementationen testen können. Wie definieren `Env` als Klasse im Trait, so dass mit dem abstrakten Type Member `Thunk` die rekursive Bezüglichkeit hergestellt werden kann. Dadurch müssen wir auch `Value` im Trait definieren.

```
trait CBN {
  type Thunk
  case class Env(map: Map[String, Thunk]) {
    def apply(key: String): Thunk = map.apply(key)
    def +(other: (String, Thunk)) : Env = Env(map+other)
  }

  def delay(e: Exp, env: Env) : Thunk
  def force(t: Thunk) : Value
```

```
  sealed abstract class Value
  case class NumV(n: Int) extends Value
  case class ClosureV(f: Fun, env: Env) extends Value

  def eval(e: Exp, env: Env) : Value = e match {
    case Id(x) => force(env(x))
    case Add(l,r) =>
      (eval(l,env), eval(r,env)) match {
        case (NumV(v1),NumV(v2)) => NumV(v1+v2)
        case _ => sys.error("can only add numbers")
      }
    case App(f,a) => eval(f,env) match {
      case ClosureV(f,cEnv) => eval(f.body, cEnv + (f.param -> delay(a,env)))
      case _ => sys.error("can only apply functions")
    }
    case Num(n) => NumV(n)
    case f@Fun(x,body) => ClosureV(f,env)
  }
}
```

Die `delay`-Funktion dient dazu, die Auswertung eines Ausdrucks zu verzögern (also einen Thunk zu erzeugen), `force` dient dazu, die "aufgeschobene" Auswertung zu erzwingen (also den Ausdruck im Thunk mit der zugehörigen Umgebung auszuwerten). Im `Id`-Fall schlagen wir den Bezeichner nach und erzwingen die Auswertung des daran gebundenen Ausdrucks, im `App`-Fall verzögern wir die Auswertung des Arguments, bevor wir es in der Umgebung an den Parameter binden.

Hier die konkrete Call-By-Name-Implementierung des `CBN`-Traits:

```
object CallByName extends CBN {
  case class Thunk(exp: Exp, env: Env)
  def delay(e: Exp, env: Env): Thunk = Thunk(e,env)
  def force(t: Thunk): CallByName.Value = {
    println("Forcing evaluation of expression "+t.exp)
    eval(t.exp,t.env)
  }
}
```

Der Typ `Thunk` wird dabei mit den Feldern `exp` und `env` definiert, diese werden durch `delay` mit dem entsprechenden Ausdruck und der aktuellen Umgebung belegt. In `force` werden die Felder ausgelesen und der Ausdruck mit der hinterlegten Umgebung ausgewertet.

Call-By-Need

Während das Argument unter Call-By-Value immer genau ein Mal ausgewertet wird, findet die Auswertung unter Call-By-Name für jede Verwendung statt. Wird das Argument einer Funktion mehrmals mehrfach weitergereicht, so kann die Anzahl der Aufrufe exponentiell wachsen. Die Auswertungsstrategie kann also in vielen Fällen sehr ineffizient sein.

Eine bessere Alternative ist die Strategie *Call-By-Name*. Dabei wird nach der ersten Auswertung des Arguments das Ergebnis durch *Caching* gespeichert, so dass bei mehrfacher Auswertung das abgespeicherte Ergebnis verwendet werden kann und Argumente höchstens ein Mal ausgewertet werden.

```
object CallByNeed extends CBN {
  case class MemoThunk(e: Exp, env: Env) {
    var cache: Value = _
  }
```

```
  type Thunk = MemoThunk
  def delay(e: Exp, env: Env): MemoThunk = MemoThunk(e,env)
  def force(t: Thunk): CallByNeed.Value = {
    if (t.cache != null)
      println ("Reusing cached value "+t.cache+" for expression "+t.e)
    else {
      println("Forcing evaluation of expression: "+t.e)
      t.cache = eval(t.e, t.env)
    }
    t.cache
  }
}
```

Wir implementieren den Typ `Thunk` wieder als mit den Feldern `exp` und `env`, diesmal aber mit einer zusätzlichen Variablenmitglied `cache`, in dem das Ergebnis der Auswertung hinterlegt werden kann. Bei der Auswertung mit `force` wird überprüft, ob das `cache`-Feld instanziiert wurde. Falls ja, kann das Ergebnis direkt von dort übernommen werden, falls nein, wird der Ausdruck mit der gespeicherten Umgebung ausgewertet und das Ergebnis in `cache` hinterlegt, bevor es ausgegeben wird.

Anhand der `print`-Ausgaben lässt sich erkennen, dass etwa beim folgenden Aufruf drei Mal auf den Cache zugegriffen wird:

```
CallByNeed.eval(App(Fun("x", Add(Add("x","x"),Add("x","x"))), Add(2,2,)))
```

Rekursive Bindings (RCFAE)

In FAE kann Rekursion zwar kodiert werden, im Gegensatz zu F1-WAE ist es aber nicht möglich, dass eine Funktion sich selbst in ihrem Rumpf aufruft:

```
val forever = wth("forever", Fun("x", App("forever","x")), App("forever",42))
```

Die Auswertung dieses Ausdrucks würde einen Fehler liefern, da der Bezeichner `"forever"` im Rumpf der Funktion nicht gebunden ist, sondern nur im dritten Ausdruck von `wth`.

Aus diesem Grund ist es auch nicht möglich, mit einem Sprachkonstrukt `If0` rekursive Funktionen mit Abbruchbedingung zu definieren:

```
val facAttempt =
  wth("fac",
      Fun("n", If0("n", 1, Mul("n", App("fac", Add("n",-1))))),
      App("fac",4))

// With fac = (n => If (n==0) 1 Else n*fac(n-1)):
//   fac(4)
```

Um Rekursion in dieser Form erzeugen zu können, brauchen wir ein Konstrukt, mit dem rekursive Bindungen möglich sind. Dazu führen wir das (analog zur Scheme-Funktion benannte) `Letrec`-Konstrukt ein, das die gleiche Form wie `With` bzw. `wth` hat, aber rekursive Bindings ermöglichen soll. Zudem erweitern wir die Sprache um ein `If0`-Konstrukt, um Abbruchbedingungen für rekursive Funktionen geschickt formulieren zu können:

```
case class If0(cond: Exp, tBranch: Exp, eBranch: Exp) extends Exp
case class Letrec(x: String, xDef: Exp, body: Exp) extends Exp

def eval(e: Exp, env: Env) : Value = e match {
  // ...
  case If0(c,t,f) => eval(c,env) match {
    case NumV(0) => eval(t,env)
```

```
    case NumV(_) => eval(f,env)
    case _ => sys.error("Can only check if number is zero")
  }
  case Letrec(i,v,b) => // ???
}
```

Es stellt sich aber die Frage, wie wir im `Letrec`-Fall vorgehen sollen. Zuerst müssen wir `xDef` auswerten, handelt es sich dabei um eine Funktion, so ist das Ergebnis der Auswertung natürlich ein Closure. Die Umgebung im Closure enthält aber keine Bindung für den Funktionsnamen, der ja im Rumpf der Funktion auftritt.

Selbst wenn man die Umgebung im Closure um eine Bindung für den Funktionsnamen erweitert, würde das nur einen rekursiven Aufruf ermöglichen, dann wäre der Funktionsname im Rumpf bereits wieder nicht gebunden. Für eine unbegrenzte Rekursionstiefe müsste für die Umgebung gelten: `env = Map("fac" -> ClosureV(Fun("n",...), env))`, sie müsste sich also zirkulär selbst referenzieren.

Eine Möglichkeit, um zirkuläre Strukturen zu definieren, ist durch Objektreferenzen (bspw. durch zwei Instanzen, die gegenseitig auf sich verweisen). Hierzu ist Mutation notwendig, es wird die erste Objektinstanz mit Null-Pointer erzeugt, dann die zweite Objektinstanz mit Pointer auf die erste, zuletzt wird der Pointer im ersten Objekt mutiert und auf das zweite gesetzt.

Wir legen eine entsprechende Datenstruktur in einem Objekt `Values` an. Diese besteht aus einem Trait `ValueHolder`, das durch die Klassen `Value` und `ValuePointer` implementiert wird. Instanzen von `Value` sind dabei selbst Werte, Instanzen von `ValuePointer` verweisen auf `Value`-Instanzen. Die `Value`-Subklassen `NumV` und `ClosureV` kennen wir bereits, der Umgebungstyp `Env` ist nun nicht mehr eine Map von `String` nach `Value` sondern von `String` nach `ValueHolder`. Damit diese zirkuläre Definition möglich ist (`ValueHolder -> Value -> ClosureV -> Env -> ValueHolder`), müssen die Definitionen innerhalb eines Objekts liegen.

```
object Values {
  trait ValueHolder {
    def value: Value
  }
  sealed abstract class Value extends ValueHolder {
    def value = this
  }
  case class ValuePointer(var v: Value) extends ValueHolder {
    def value = v
  }
  case class NumV(n: Int) extends Value
  case class ClosureV(f: Fun, env: Env) extends Value
  type Env = Map[String,ValueHolder]
}
```

Anschließend importieren wir die Definitionen:

```
import Values._
```

Die `eval`-Funktion ändert sich an folgenden Stellen:

```
def eval(e: Exp, env: Env) : Value = e match {
  // ...
  case Id(x) => env(x).value
  // ...
  case Letrec(i,v,b) =>
    val vp = ValuePointer(null)
    val newEnv = env+(i -> vp)
    vp.v = eval(v,newEnv)
    eval(b,newEnv)
```

```
}
```

Im `Letrec`-Fall definieren wir erst den `ValuePointer` `vp`, den wir mit einem Verweis auf `null` initialisieren. Wir erweitern die aktuelle Umgebung mit der Bindung des Identifiers `i` auf `vp`. Dann mutieren wir den `ValuePointer` so, dass er das Auswertungsergebnis des zu bindenden Wertes in der neuen Umgebung referenziert (Kreisschluss!). Nach dieser Mutation ist mit der neuen Umgebung die Auswertung möglich.

Am Beispiel der `forever`-Funktion geschieht folgendes:

```
eval( Letrec("forever", Fun("x",App("forever","x")), App("forever",42)), Map() )
// ~~>
eval( App("forever",42), Map("forever" -> vp) )
// vp.v = ClosureV( Fun("x",App("forever","x")), Map("forever" -> vp) )
```

Bei der Auswertung von `App("forever",42)` wird im `Id`-Fall der Identifier `"forever"` in der Umgebung nachgeschlagen und auf das `value`-Feld des Ergebnisses zugegriffen. Dabei handelt es sich um den oben auskommentierten Closure. Bei der Auswertung des Funktionsrumpfes wird die Umgebung aus dem Closure erweitert, in dieser ist `"forever"` wieder an `vp` gebunden. Bei einem rekursiven Aufruf wird also wieder `vp.value` ausgelesen, etc.

Damit ist eine unbegrenzte Rekursionstiefe möglich.

Mutation (BCFAE)

Die Sprache FAE (auch inkl. `Letrec`) ist eine *rein funktionale* Sprache, also eine Sprache ohne Mutation und Seiteneffekte. In dieser Art von Sprache lassen sich Programme und deren Auswertung besonders leicht nachvollziehen und es liegt *Referential Transparency* vor.

Referential Transparency bedeutet, dass alle Aufrufe einer Funktion mit dem gleichen Argument überall durch das (identische) Ergebnis des Aufrufs ersetzt werden können, ohne die Bedeutung des Programms zu verändern.

Besitzen Funktionen Seiteneffekte (etwa Print-Befehle oder Mutationen), so ist dies nicht der Fall, denn durch das Ersetzen des Funktionsaufrufs mit dem Ergebnis gehen jegliche Seiteneffekte verloren.

Eine Form von Mutation ist das Mutieren von Variablen, also das Überschreiben des Wertes einer Variable:

```
var x = 1
x = 2
```

Eine andere Form ist die mutierbarer Datenstrukturen, bspw. Arrays, in denen einzelne Werte überschrieben werden können. Wir wollen die einfachste denkbare Form einer solchen mutierbaren Datenstruktur unserer Sprache hinzufügen, nämlich eine Datenstruktur mit genau einem Wert, die wir als *Box* bezeichnen.

Box-Container

Eine Box entspricht einem Array der Länge 1, ist also ein Datencontainer für genau einen Wert. Um Boxes zu implementieren, führen wir die folgenden Sprachkonstrukte ein:

```
case class NewBox(e: Exp) extends Exp
case class SetBox(b: Exp, e: Exp) extends Exp
case class OpenBox(b: Exp) extends Exp
case class Seq(e1: Exp, e2: Exp) extends Exp
```

Wir müssen Boxen instanziieren, beschreiben und auslesen bzw. dereferenzieren können, zudem brauchen wir eine Möglichkeit, um zu Sequenzieren, also zwei Ausdrücke nacheinander auszuwerten (damit ein Ausdruck neben einer Mutation auch eine Berechnung durchführen kann). Da der Wert einer Box mutiert werden kann, spielt die Auswertungsreihenfolge von Unterausdrücken nun eine entscheidende Rolle.

Wir implementieren den `Box`-Container aus pädagogischen Gründen nicht durch Mutation in der Meta-Sprache, sondern bleiben weiterhin bei einem funktionalen Interpreter.

```
val ex1 = wth("b", NewBox(0), Seq( SetBox("b", Add(1,OpenBox("b"))), OpenBox("b")))
/* Should evaluate to 1.
With b = NewBox(0):
  SetBox(b <- 1+OpenBox(b));
    OpenBox(b)
 */
```

Die Implementierung von `Seq` stellt uns vor eine Herausforderung, die Reihenfolge der rekursiven `eval`-Aufrufe in unserem Interpreter spielt nämlich keine Rolle, da diese Funktionsaufrufe keine Effekte haben (und auch nicht haben sollen). Wir müssen also unseren Interpreter abändern, so dass nach der Auswertung sowohl das Ergebnis, als auch durchgeführte Mutationen zurückgegeben werden, damit wir beim Auswerten des zweiten Programmabschnitts die Effekte des ersten Programmabschnitts berücksichtigen können.

Hierzu ist es aber nicht ausreichend, `(Value,Env)` als Rückgabetyp zu wählen, wie das folgenden Beispiel zeigt:

```
val ex2 = wth("a", NewBox(1),
            wth("f", Fun("x", Add("x", OpenBox("a"))),
              Seq(SetBox("a",2), App("f",5))))
/* Should evaluate to 7.
With b = NewBox(1):
  With f = (x => x+OpenBox(b):
    SetBox(a, 2);
    f(5)
 */
```

Bei der Auswertung von `f` wird die Umgebung aus dem zu `f` gehörigen Closure verwendet. In dieser Umgebung steht in der an `"a"` gebundenen Box der Wert `1`, nicht `2`. Environments dienen zur Umsetzung von lexikalischem Scoping, sie werden entsprechend der Programmstruktur rekursiv in Unterausdrücke weitergereicht. Für Mutation ist aber die Auswertungsreihenfolge und nicht die syntaktische Struktur des Programms entscheidend, insofern sind Environments für die Implementation dieses neuen Features ungeeignet.

Stattdessen benötigen wir eine zweite Datenstruktur, die wir als zusätzliches Argument bei der Auswertung übergeben und in evtl. modifizierter Form nach der Auswertungs ausgeben. Die neue Datenstruktur besitzt einen ganz anderen Datenfluss als die Umgebung, sie wird von Auswertungsposition zu Auswertungsposition gereicht und nicht wie die Umgebung im AST immer nur nach unten weitergegeben.

Store und Adressen

Wir verwenden wieder unsere alten Definitionen von `Value` und `Env` und führen die Typen `Address` und `Store` ein. Zusätzlich erweitern wir `Value` um den Fall `AddressV`:

```
sealed abstract class Value
type Env = Map[String,Value]
case class NumV(n: Int) extends Value
case class ClosureV(f: Fun, env: Env) extends Value

type Address = Int
case class AddressV(a: Address) extends Value
type Store = Map[Address,Value]
```

Adressen sind Integers und dienen als Referenzen (“Pointer”) für Box-Instanzen. In der `Store`-Map werden die aktuellen Werte der Box-Instanzen hinterlegt. Um Identifier an Boxen binden zu können, müssen wir Boxen als `Value` repräsentieren können, zu diesem Zweck dient `AddressV`.

Um neue Adressen zu erhalten, inkrementieren wir einfach die bisher höchste Adresse um 1. Um das Entfernen alter Referenzen und die Limitierungen dieses Adressen-Systems kümmern wir uns zum jetzigen Zeitpunkt nicht.

```
var address = 0
def nextAddress : Address = {
  address += 1
  address
}
```

Hier verwendet unsere Implementation doch Mutation in der Meta-Sprache, aber nur um diese Hilfsfunktion zu vereinfachen. Alternativ wäre eine Funktion `freshAddress` denkbar, die eine neue ungenutzte Adresse erzeugt (vgl. `freshName` hier)

Interpreter

Im Interpreter ändert sich in allen Fällen deutlich, die Implementation wird im Allgemeinen aufwändiger:

```
def eval(e: Exp, env: Env, s: Store) : (Value,Store) = e match {
  case Num(n) => (NumV(n),s)
  case Id(x) => (env(x),s)
  case f@Fun(_,_) => (ClosureV(f,env),s)
  case Add(l,r) => eval(l,env,s) match {
    case (NumV(a),s1) => eval(r,env,s1) match {
      case (NumV(b),s2) => (NumV(a+b),s2)
      case _ => sys.error("Can only add numbers")
    }
    case _ => sys.error("Can only add numbers")
  }
  case App(f,a) => eval(f,env,s) match {
    case (ClosureV(Fun(p,b),cEnv),s1) => eval(a,env,s1) match {
      case (v,s2) => eval(b, cEnv+(p -> v), s2)
    }
    case _ => sys.error("Can only apply functions")
  }
  case Seq(e1,e2) =>
    eval(e2,env,eval(e1,env,s)._2)
  case NewBox(e) => eval(e,env,s) match {
    case (v,s1) =>
      val a = nextAddress
      (AddressV(a), s1+(a -> v))
  }
  case SetBox(b,e) => eval(b,env,s) match {
    case (AddressV(a),s1) => eval(e,env,s1) match {
      case (v,s2) => (v, s2+(a -> v))
    }
    case _ => sys.error("Can only set boxes")
  }
  case OpenBox(b) =>  eval(b,env,s) match {
    case (AddressV(a),s1) => (s1(a),s1)
    case _ => sys.error("Can only open boxes")
  }
}
```

Der `Num`-, `Id`- und `Fun`-Fall entsprechen der umgebungsbasierten FAE-Implementierung bis auf den neuen Store, der zusätzlich (mit dem Ergebnis zusammen in einem Tupel) ausgegeben wird. Im `Add`- und `Mul`-Fall

müssen wir aber bereits entscheiden, in welcher Reihenfolge wir den linken und rechten Unterausdruck auswerten wollen. Wir werten (wie die meisten Sprachen) erst links und dann rechts aus.

Durch die Auswertung des linken Teilausdrucks (mit aktueller Umgebung und aktuellem Store) erhalten wir ein Tupel aus `Value` und `Store`, diesen neuen `Store` verwenden wir dann bei der Auswertung des rechten Teilausdrucks, so dass potentielle Mutierungen im linken Teilausdruck bei der Auswertung des rechten Teilausdrucks berücksichtigt werden. Die Auswertung des rechten Teilausdrucks liefert wieder einen Wert und einen Store, wir geben die Summe der Zahlen und den neuesten Store als Ergebnis aus.

Auch im `App`-Fall müssen wir den Store erst in den linken Unterausdruck und dann (potentiell modifiziert) in den rechten Ausdruck reichen, der Store, der bei der Auswertung des Arguments ausgegeben wird, verwenden wir bei der Auswertung des Funktionsrumpfs.

Im `Seq`-Fall werten wir zuerst den linken Ausdruck aus und greifen mit `._2` auf den Store aus dem Ergebnis zu. Diesen nutzen wir dann bei der Auswertung des rechten Ausdrucks. Das Ergebnis des linken Teilausdrucks wird also ignoriert, es wird das Ergebnis des rechten Teilausdrucks ausgegeben.

Um eine neue Box-Instanz zu erzeugen, werten wir zuerst den Ausdruck aus, der in der Box stehen soll. Wir erhalten einen Wert `v` und einen Store `s1`, erzeugen mit `nextAddress` eine neue Adresse und geben ein Tupel aus der neuen Adresse und `s1`, erweitert um eine Bindung der neuen Adresse an `v`, aus. Im `SetBox`-Fall muss zusätzlich der Ausdruck an der ersten Stelle ausgewertet werden, um die Adresse der Box-Instanz zu erhalten und den Store mit dem neuen Wert zu aktualisieren. Im `OpenBox`-Fall wird auch erst die Adresse bestimmt, anschließend wird der an die Adresse gebundene Wert und der aktuelle Store ausgegeben.

Beim Auslesen einer Box-Instanz wird also erst in der Umgebung der Bezeichner nachgeschlagen, was einen `AddressV`-Wert liefern sollte, anschließend wird im Store nachgeschlagen, auf welchen Wert diese Adresse verweist.

Speichermanagement

Die Funktion `nextAddress`, mit der wir ungenutzte Adressen für neue Boxen erzeugen, inkrementiert einfach die Variable `address` immer weiter. Der Wert von `address` wird nach der Auswertung nicht zurückgesetzt. Während der Auswertung wird auch nicht geprüft, welche Einträge im Store noch benötigt werden und ob Adressen und die an sie gebundenen Werte entfernt werden können.

Eine Möglichkeit, nicht mehr benötigte Einträge zu entfernen, wäre ein neues Sprachkonstrukt, etwa `RemoveBox`. Damit könnte der Programmierer Box-Instanzen verwerfen, die nicht mehr benötigt werden. Wird aber ein Identifier an eine Box-Instanz gebunden und diese Box-Instanz gelöscht, so verweist der Identifier weiterhin auf eine Adresse, für die es im Store aber keinen Eintrag mehr oder sogar einen anderen, neuen Eintrag gibt.

Unter anderem durch solche *Dangling Pointers* ist Programmieren mit manuellem Speichermanagement fehleranfällig, auch wenn dadurch performantere Programme möglich sind. Aus diesem Grund verwenden viele Programmiersprachen automatisches Speichermanagement in Form von *Garbage Collection*.

Garbage Collection

Garbage Collection beruht auf der Tatsache, dass algorithmisch bestimmt bzw. approximiert werden kann, welche Speicherinhalte in der weiteren Auswertung noch benötigt werden.

Ideal wäre ein Garbage-Collection-Algorithmus, der folgendes erfüllt:

"Perfekte" Garbage Collection: Wenn die Adresse a im Store s in der weiteren Berechnung nicht mehr benötigt wird, so wird der Eintrag für a aus s entfernt.

Die Fragestellung, ob ein Store-Eintrag in der weiteren Berechnung noch benötigt wird, ist jedoch unentscheidbar, was aus der Unentscheidbarkeit des Halteproblems und dem Satz von Rice folgt. Wird bspw. eine Funktion f aufgerufen und danach auf eine Adresse zugegriffen, so wird die Adresse nur benötigt, wenn f terminiert. Für perfekte Garbage Collection müsste also das Halteproblem entscheidbar sein.

Es kann jedoch die Menge der noch benötigten Adressen approximiert werden. Approximinieren bedeutet dabei, das es Adressen gibt, für die keine Entscheidung möglich ist oder die falsch eingeordnet werden. Garbage Collection wird dabei so gestaltet, dass nur eine Art von Fehler geschieht, nämlich dass Daten unnötig/fälschlicherweise im Speicher gehalten werden aber nie fälschlicherweise verworfen werden (*Soundness*).

Reachability von Speichereinträgen hat sich als eine geeignete Approximation für die Zwecke von Garbage Collection herausgestellt und wird in vielen Algorithmen benutzt, um nicht mehr benötigte Einträge zu bestimmen.

Erreichbarkeit/Reachability: Eine Adresse ist *erreichbar*, wenn sie sich in der aktuellen Umgebung (inkl. Unterumgebungen in Closures, usw.) befindet, oder wenn es einen Pfad von Verweisen aus der aktuellen Umgebung zu der Adresse gibt.

Garbage Collection entspricht also dem Erreichbarkeitsproblem in einem gerichteten Graphen. Voraussetzung ist dabei, dass alle nicht erreichbaren Adressen im Rest der Berechnung nicht benötigt werden. Das wäre bspw. nicht der Fall, wenn durch Pointer-Arithmetik auf beliebige Adressen zugegriffen werden kann.

Da unsere Auswertungsfunktion rekursiv ist, reicht es nicht, nur die aktuelle Umgebung zu betrachten. Es muss für jede Instanz der `eval`-Funktion auf dem Call-Stack die zugehörige Umgebung berücksichtigt werden, da beim Aufstieg aus den rekursiven Aufrufen wieder die auf dem Stack abgelegten Umgebungen verwendet werden.

Mark and Sweep

Die meisten einfachen Garbage-Collection-Algorithmen auf Basis von Reachability bestehen aus den zwei Phasen *Mark* und *Sweep*. Im ersten Schritt werden alle Adressen markiert, die noch benötigt werden, im zweiten Schritt werden dann alle nicht markierten Adressen entfernt.

Die folgende Funktion führt Mark-And-Sweep Garbage-Collection in BCFAE für eine Umgebung `env` auf dem Store durch:

```
def gc(env: Env, store: Store) : Store = {

  def allAddrInVal(v: Value) : Set[Address] = v match {
    case NumV(_) => Set()
    case ClosureV(_,env) => allAddrInEnv(env)
    case AddressV(a) => Set(a)
  }

  def allAddrInEnv(env: Env) : Set[Address] =
    env.values.map{allAddrInVal}.fold(Set())(_++_)

  def mark(seed: Set[Address]) : Set[Address] = {
    val newAddresses = seed.flatMap(a => allAddrInVal(store(a)))
    if (newAddresses.subsetOf(seed)) seed else mark(seed++newAddresses)
  }

  val marked = mark(allAddrInEnv(env)) // mark

  store.filter{ case (a,_) => marked(a) } // sweep

}
```

Die Funktion `allAddrInEnv` sammelt alle erreichbaren Adressen in einer Umgebung, indem `allAddrInVal` auf alle Werte in der Umgebung aufgerufen wird und die entstehenden Mengen alle vereinigt werden. Die Funktion `mark` erhält eine Adressmenge `seed` und liefert die Menge aller Adressen, die von den Adressen in `seed` aus erreicht werden können. Werden dabei keine neuen Adressen gefunden, so wird die Menge `seed` ausgegeben. Ansonsten wird `mark` rekursiv aufgerufen, dabei wird die Menge um die neu gefundenen Adressen

erweitert. Sie wird also schrittweise erweitert, bis keine neuen Adressen mehr gefunden werden. `gc` bestimmt erst die Menge der markierten Adressen und filtert dann den Store, so dass unmarkierte Adressen entfernt werden.

Das folgende Beispiel zeigt die Funktionsweise des Algorithmus, `gc` wird mit einer Umgebung aufgerufen, in der auf der rechten Seite die Adresse `5` vorkommt. An der Adresse `5` steht im Store eine Closure, in dem wiederum die Adresse `3` auftritt, an der Adresse `3` wird auf die Adresse `1` verwiesen. Die Adressen `2` und `5` sind nicht erreichbar und sind dementsprechend im Ergebnis-Store nicht mehr vorhanden.

```
val testEnv = Map("a" -> AddressV(5))
val testStore = Map(
  1 -> NumV(0),
  2 -> NumV(0),
  3 -> AddressV(1),
  4 -> AddressV(2),
  5 -> ClosureV(Fun("x","x"), Map("y" -> AddressV(3))))

// addresses 2 and 4 cannot be reached, are removed by GC
assert(gc(testEnv,testStore) ==
  Map(5 -> ClosureV(Fun("x","x"), Map("y" -> AddressV(3))),
      3 -> AddressV(1),
      1 -> NumV(0)))
```

Moving und Non-Moving GC

Durch wiederholtes Anlegen und Entfernen von Speichereinträgen kann es zu einer starken Fragmentierung des Speichers kommen, die belegten Speicherzellen sind dann evtl. stark verteilt und der Speicher ist "lückenhaft" befüllt.

Diese Fragmentierung erschwert zum einen die Speicherzuweisung, da größere "Datenblöcke" evtl. nicht am Stück gespeichert werden können und zerlegt werden müssen, zum anderen verschlechtert sich die Performanz, da die Speichernutzung weniger effizient wird (freier Speicher ist verteilt und kann dadurch evtl. nicht genutzt werden, es müssen mehr Adressen im Speicher gehalten werden, zusammengehörige Daten werden nicht automatisch gemeinsam in den Cache geladen).

Um Fragmentierung zu reduzieren, müssen bei der Garbage Collection Speichereinträge verschoben ("zusammengerückt") werden, sodass die Speicherbelegung möglichst dicht bzw. kompakt bleibt. Hierbei spricht man von *Moving* Garbage Collection. Dabei werden nicht nur die Daten verschoben, sondern es müssen auch alle Referenzen mit der neuen Speicheradresse aktualisiert werden.

Bei Mark and Sweep werden die Daten nicht verschoben, der Algorithmus ist eine Form von *Non-Moving* Garbage Collection. Er führt allgemein über die Laufzeit hinweg zu zunehmender Fragmentierung.

Ein Beispiel für Moving Garbage Collection ist *Semi-Space Garbage Collection*. Dabei wird der Speicher in zwei Hälften geteilt, wobei während der Allokation nur eine Hälfte des Speichers verwendet wird. Ist diese voll, so wird Garbage Collection durchgeführt: Die noch benötigten Einträge werden markiert, anschließend werden alle markierten Einträge in die freie Speicherhälfte kopiert, wodurch der Speicher wieder dicht belegt wird. Zuletzt werden alle Einträge in der vollen Speicherhälfte gelöscht. Beim nächsten GC-Zyklus wird das Verfahren mit umgekehrten Rollen wiederholt.

Vorteil ist, dass bei jedem GC-Zyklus der gesamte Speicher defragmentiert wird, das Problem der steigenden Fragmentierung über Zeit ist also behoben. **Nachteile** sind die hinzukommenden Kopieroperationen und größere Anzahl an Löschoperationen, die Aktualisierung der Referenzen und die dazu notwendigen Suchoperationen, sowie die (im Worst Case) Halbierung des verfügbaren Speicherplatzes.

Weitere Begriffe

- **Generational GC:** Es kann empirisch belegt werden, dass in den meisten Anwendungen die Objekte, die bei einem GC-Zyklus dereferenziert werden können, tendenziell sehr "jung" sind, also erst vor kurzer Zeit angelegt wurden. Bei Objekten, die sich schon sehr lange im Speicher befinden, werden viel wahrscheinlicher noch benötigt als Objekte, die erst kürzlich angelegt wurden. Bei *Generational Garbage Collection* macht man sich diese Tatsache zunutze, indem die Objekte nach ihrem Alter aufgeteilt werden und im Speicherbereich für jungen Objekte öfter Garbage Collection durchgeführt wird als im Speicherbereich für alte Objekte. Somit kann Speicherplatz effizienter freigegeben werden, da gezielt die Objekte betrachtet werden, die am ehesten dereferenziert werden können.
- **"Stop the World"-Phänomen:** Während Garbage Collection durchgeführt wird, kann eine Anwendung i.A. nicht weiterlaufen, denn der GC-Algorithmus wäre nicht mehr sicher, wenn während der Ausführung des GC-Algorithmus weiter Adressen angelegt und Referenzen geändert werden. Stattdessen muss der Anwendungsprozess aufgeschoben werden, bis der GC-Zyklus vollendet ist. In den meisten Fällen kann dies unbemerkt geschehen, aber bei interaktiven Programmen und Echtzeit-Anwendungen wird die Garbage Collection und das damit verbundene Aussetzen des Programms evtl. durch Ruckeln oder kurzes "Hängen" bemerkbar. Im besten Fall ist das für einen Nutzer leicht störend, im schlimmsten Fall hat die verzögerte Reaktion aber weitreichende Folgen, weshalb automatisches Speichermanagement für Programme mit extrem hohen Ansprüchen an die Reaktionsfähigkeit und Zuverlässigkeit ungeeignet sein kann.
- **Reference Counting:** Eine andere Form des automatischen Speichermanagements, die nicht auf Erreichbarkeit von Objektinstanzen beruht, ist *Reference Counting*. Dabei wird zusammen mit jeder Objektinstanz ein Feld angelegt, in dem die Anzahl der Referenzen auf das Objekt gehalten wird. Ist diese Anzahl 0, so kann das Objekt dereferenziert werden. Bei jeder Änderung der Referenzen müssen die Felder in den betroffenen Objekten aktualisiert werden. Im Gegensatz zu Garbage Collection muss die Anwendung nicht mehr unterbrochen werden, Objekte können gelöscht werden, sobald ihr Counter 0 beträgt. Gibt es jedoch Referenzzyklen, so werden Objekte, die evtl. nicht mehr erreichbar sind, dennoch im Speicher gehalten. Deshalb wird das Verfahren typischerweise mit Zyklendetektion kombiniert, damit solche Strukturen erkannt und die entsprechenden Objekte dereferenziert werden.

Interpreter mit Speichermanagement

Um Garbage Collection in den BCFAE-Interpreter zu integrieren, ergänzen wir Werte um einen "Markierungszustand" und verwenden eine neue Definition für `Store`:

```
sealed abstract class Value {
  var marked : Boolean = false
}

trait Store {
  def malloc(stack: List[Env], v: Value) : Int
  def update(i: Int, v: Value) : Unit
  def apply(i: Int) : Value
}
```

Dabei fügt `malloc` dem Store einen Wert hinzu (wobei falls notwendig GC betrieben wird) und liefert die gewählte Adresse zurück, `update` mutiert den Wert an der Adresse `i` im Store und `apply` liest den Wert an der Adresse `i` aus. Wir verwenden jetzt also auch Mutation in der Meta-Sprache, um den Interpreter samt Speichermanagement zu implementieren. Der Store wird also nicht mehr von Funktionsaufruf zu Funktionsaufruf gereicht, sondern ist eine globale Datenstruktur, auf die von überall zugegriffen werden kann.

Unser Interpreter ähnelt dadurch wieder stärker dem umgebungsbasierten FAE-Interpreter:

```
def eval(e: Exp, stack: List[Env], store: Store) : Value = e match {
  case Num(n) => NumV(n)
```

```
  case Id(x) => stack.head(x)
  case Add(l,r) => (eval(l,stack,store),eval(r,stack,store)) match {
    case (NumV(a),NumV(b)) => NumV(a+b)
    case _ => sys.error("Can only add numbers")
  }
  case Mul(l,r) => // analogous to Add
  case f@Fun(_,_) => ClosureV(f,stack.head)
  case If0(c,t,f) => eval(c,stack,store) match {
    case NumV(0) => eval(t,stack,store)
    case NumV(_) => eval(f,stack,store)
    case _ => sys.error("Can only check if number is zero")
  }
  case App(f,a) => eval(f,stack,store) match {
    case ClosureV(Fun(p,b),cEnv) =>
      eval(b, cEnv+(p -> eval(a,stack,store))::stack, store)
    case _ => sys.error("Can only apply functions")
  }
  case Seq(e1,e2) => eval(e1,stack,store); eval(e2,stack,store)
  case NewBox(e: Exp) =>
    val a = store.malloc(stack,eval(e,stack,store))
    AddressV(a)
  case SetBox(b: Exp, e: Exp) => eval(b,stack,store) match {
    case AddressV(a) =>
      val v = eval(e,stack,store)
      store.update(a,v)
      v
    case _ => sys.error("Can only set boxes")
  }
  case OpenBox(b: Exp) => eval(b,stack,store) match {
    case AddressV(a) => store.apply(a)
    case _ => sys.error("Can only open boxes")
  }
}
```

Statt einer Umgebung verwendet der Interpreter nun eine Liste aller im Call-Stack auftretenden Umgebungen als Parameter. Der Kopf der Liste ist die aktive Umgebung des aktuellen Funktionsaufrufs, die weiteren Elemente sind die Umgebungen der "darüberliegenden" Aufrufe in der Rekursionsstruktur, als die auf dem Call-Stack abgelegten Umgebungen. Im `App`-Fall wird die Umgebungsliste erweitert, indem vorne an die Liste eine neue Umgebung angehängt wird, die zusätzlich die Bindung des Parameters enthält. Im `Id`-Fall wird der Bezeichner in der obersten/ersten Umgebung im Stack nachgeschlagen.

Im `NewBox`-, `SetBox`- und `OpenBox`-Fall werden jeweils die Store-Funktionen `malloc`, `update` und `apply` benutzt. Es muss der Stack anstelle der Umgebung als Parameter von `eval` verwendet werden, damit dieser im `NewBox`-Fall an die `malloc`-Funktion überreicht werden kann – für GC und die Suche nach ungenutzten Adressen werden nämlich alle Umgebungen im Stack benötigt.

Ohne GC

Eine sehr einfache Implementation der abstrakten `Store`-Klasse für Speichermanagement ohne Garbage Collection sieht folgendermaßen aus:

```
class StoreNoGC(size: Int) extends Store {
  val memory = new Array[Value](size)
  var nextFreeAddr: Int = 0
  def malloc(stack: List[Env], v: Value) : Int = {
    val a = nextFreeAddr
```

```
    if (a >= size) sys.error("Out of memory!")
    nextFreeAddr += 1; update(a,v); a
  }
  def update(i: Int, v: Value) : Unit = memory.update(i,v)
  def apply(i: Int) : Value = memory(i)
}
```

Der Store ist hierbei durch ein Array implementiert, um eine freie Adresse zu finden wird die mutierbare Variable `nextFreeAddr` verwendet und nach jeder neuen Zuweisung inkrementiert. Die Größe `size` wird bei der Instanziierung gewählt und legt fest, wie groß das Array ist, also wie viele Adressen es gibt. Sind alle Indices des Array belegt worden, so ist der Speicher voll und es wird eine Fehlermeldung ausgegeben. Es wird keinerlei Garbage Collection betrieben, um Ressourcen freizugeben.

Die `eval`-Aufrufe im folgenden Programm verursachen also eine Fehlermeldung, wenn bei der Evaluation von `ex1` mindestens eine Box angelegt wird:

```
val store = new StoreNoGC(2)
val stack = List[Env]()
eval(ex1,stack,store); eval(ex1,stack,store); eval(ex1,stack,store)
```

Auch ein Testprogramm, in dem mehr als zwei Boxen instanziiert werden, würde einen Fehler liefern.

Mit GC

Wir implementieren nun die abstrakte `Store`-Klasse mit "Mark & Sweep"-Garbage-Collection. Der Store wird dabei wieder mit einer Größe `size` instanziiert, die die Anzahl der Adressen bestimmt. Wir ergänzen eine Variable `free`, in der die Anzahl ungenutzter Adressen gehalten wird. Gibt es bei der Speicherallokation keine freien Adressen mehr, so wird Garbage Collection betrieben. Ist auch danach keine Adresse frei, so wird eine Fehlermeldung ausgegeben. Der verwendete "Mark & Sweep"-Algorithmus ähnelt dem im vorherigen Kapitel aufgeführten stark:

```
class MarkAndSweepStore(size: Int) extends Store {
  val memory = new Array[Value](size)
  var free : Int = size
  var nextFreeAddr : Int = 0
  def malloc(stack: List[Env], v: Value) : Int = {
    if (free <= 0) gc(stack)
    if (free <= 0) sys.error("Out of memory!")
    while (memory(nextFreeAddr) != null) {
      nextFreeAddr += 1
      if (nextFreeAddr == size) nextFreeAddr = 0
    }
    update(nextFreeAddr,v); free -= 1; nextFreeAddr
  }
  def update(i: Int, v: Value) : Unit = { memory(i) = v }
  def apply(i: Int) : Value = memory(i)

  // Mark & Sweep GC:
  def allAddrInVal(v: Value) : Set[Int] = v match {
    case NumV(_) => Set()
    case AddressV(a) => Set(a)
    case ClosureV(_,env) => allAddrInEnv(env)
  }
  def allAddrInEnv(env: Env) : Set[Int] = {
    env.values.map(allAddrInVal).fold(Set())(_++_)
  }
  def mark(seed: Set[Int]) : Unit = {
```

```
    seed.foreach(memory(_).marked = true)
    val newAddresses =
      seed.flatMap(a => allAddrInVal(memory(a))).filter(!memory(_).marked)
    if (newAddresses.nonEmpty) {
      mark(newAddresses)
    }
  }
  def sweep() : Unit = {
    memory.indices.foreach(
      i => if (memory(i) == null) {}
      else if (memory(i).marked) memory(i).marked = false
      else { memory(i) = null; free += 1 }
    )
  }
  def gc(stack: List[Env]) : Unit = {
    mark(stack.map(allAddrInEnv).fold(Set())(_++_))
    sweep()
  }
}
```

Gibt es bei der Allokation noch (oder nach der Garbage Collection) freie Adressen, so wird das Array linear durchsucht, bis ein leeres Feld gefunden wird. Hier wird dann der Wert eingetragen, `free` wird dekrementiert und die Adresse wird ausgegeben.

Die Markierung der noch erreichbaren Adressen ist nicht mehr durch eine Menge repräsentiert, sondern durch das Feld `marked` in jedem `Value`. Die `sweep`-Funktion ersetzt nicht markierte Werte im Store durch `null` (wobei `free` inkrementiert wird) und setzt die Markierung aller Werte auf `false` zurück.

Meta- und syntaktische Interpretation

Jede Sprachsemantik einer Programmiersprache lässt sich in einer Meta-Sprache auf verschiedene Arten implementieren, dabei ist die Unterscheidung zwischen *Metainterpretation* und *syntaktischer Interpretation* von besonderer Wichtigkeit.

Metainterpretation bezeichnet die Implementierung eines Sprachfeatures durch das entsprechende Feature in der Hostsprache, *syntaktische Interpretation* hingegen die Implementierung eines Features durch Reduktion auf primitivere Sprachkonstrukte der Hostsprache. Metainterpretation ist (falls überhaupt möglich) leichter zu implementieren, erlaubt aber keine Anpassung eines Sprachkonstrukts gegenüber der Implementation in der Hostsprache. Syntaktische Interpretation erlaubt eine andere, selbst festgelegte Implementation und die Kontrolle darüber, wobei aber ein umfangreiches Verständnis des Sprachkonstrukts notwendig ist, um dieses mit einfacheren Mitteln selbst zu implementieren.

In unserer Sprache FAE ist bspw. Addition durch Metainterpretation implementiert, wir verwenden im Interpreter die Additionsfunktion von Scala und delegieren damit dieses Feature einfach an die Hostsprache. Dementsprechend besitzt Addition in unserer Sprache die gleichen Einschränkungen und Eigenschaften wie Addition in der Scala. Auch die maximale Tiefe rekursiver Programme oder das Speichermanagement wird nicht durch unsere Implementierung festgelegt, sondern durch Scala, da wir Rekursion in Scala für unseren Interpreter nutzen und implizit das Speichermanagement von Scala übernehmen.

Andere Sprachfeatures werden hingegen nicht durch das entsprechende Feature in der Hostsprache umgesetzt, z.B. Identifier (inkl. Scoping) oder Closures. Hier liegt syntaktische Interpretation vor. Beim Entwerfen des Interpreters muss man sich also bewusst sein, welche Verhaltensweisen und Einschränkungen mit den Features der Hostsprache einhergehen und entscheiden, welche Features man selbst implementieren und welche man "weiterreichen" (d.h. an die Hostsprache delegieren) möchte.

Wir könnten in FAE aber auch mehr Metainterpretation verwenden, indem wir etwa die Funktionen durch

Funktionen der Metasprache umsetzen:

```
sealed trait Exp
case class Num(n: Int) extends Exp
case class Id(x: String) extends Exp
case class Add(lhs: Exp, rhs: Exp) extends Exp
case class Fun(f: Exp => Exp) extends Exp
case class App(fun: Exp, arg: Exp) extends Exp
```

Eine solche Repräsentation von Funktionen der Objektsprache durch Funktionen der Metasprache nennt man *Higher-Order Abstract Syntax (HOAS)*.

Der Interpreter wird nun extrem einfach, aber die Kontrolle über das Verhalten von Identifiern und Bindungen (also bspw. Scoping) geht verloren.

```
def eval(e: Exp) : Exp = e match {
  case Id(x) => sys.error("Unbound identifier: "+x)
  case Add(l,r) => (eval(l),eval(r)) match {
    case (Num(a),Num(b)) => Num(a+b)
    case _ => sys.error("Can only add numbers")
  }
  case App(f,a) => f match {
    case Fun(f) => eval(f(eval(a)))
    case _ => sys.error("Can only apply functions")
  }
  case _ => e
}
```

Es ist auch möglich, Closures durch Metainterpretation umzusetzen:

```
sealed abstract class Value
type Env = Map[String, Value]
case class NumV(n: Int) extends Value
case class FunV(f: Value => Value) extends Value

def eval(e: Exp) : Env => Value = e match {
  case Num(n: Int) => (env) => NumV(n)
  case Id(x) => env => env(x)
  case Add(l,r) => { (env) =>
    (eval(l)(env),  eval(r)(env)) match {
      case (NumV(v1),NumV(v2)) => NumV(v1+v2)
      case _ => sys.error("Can only add numbers")
    }
  }
  case Fun(param,body) => (env) => FunV( (v) => eval(body)(env + (param -> v)))
  case App(f,a) => (env) => (eval(f)(env), eval(a)(env)) match {
    case (FunV(g),arg) => g(arg)
    case _ => sys.error("Can only apply functions")
  }
}
```

Anstelle von Closures bestehend aus einer Funktion und der zugehörigen Umgebung liefert die Auswertung von Funktionen eine `FunV`-Instanz mit einer Scala-Funktion, die die Auswertung des Rumpfes mit der korrekten Umgebung (vom Zeitpunkt, zu dem der `Fun`-Ausdruck ausgewertet wird, entsprechend der Closure-Umgebung) durchführt. Die Closures der Objektsprache sind also durch die Closures der Metasprache implementiert.

Durch das zusätzliche Auslagern des Umgebungsparameters aus `eval` durch Currying ist dieser Interpreter kompositional, d.h. alle rekursiven Aufrufe von `eval` sind auf Unterausdrücken des aktuellen Ausdrucks.

Dadurch lässt sich Programmäquivalenz in der Objektsprache leicht durch Äquivalenz in der Metasprache beweisen. Außerdem lässt sich der Interpreter auch im Visitor-Stil implementieren.

Es könnten auch mehr Sprachkonstrukte von FAE durch syntaktische Interpretation umgesetzt werden, bspw. könnte man Zahlen als Sequenz von Ziffern anstelle von Scala-`Int`s repräsentieren. Wir werden noch Implementation kennenlernen, die nicht mehr von Scalas Speichermanagement und Higher-Order-Funktionen abhängen.

Objekt-Algebren

Objekt-Algebren (*Object Algebras*) sind eine Abstraktion, die eng mit algebraischen Datentypen und Church-Kodierungen verwandt ist. Zudem haben sie einige Gemeinsamkeiten mit dem Visitor Pattern.

Sie sind ein mächtiger und teilweise auch effizienter Mechanismus zur Modularisierung von Programmen und ein sehr junges Forschungsgebiet, zu dem es erst seit etwa 2012 Veröffentlichungen gibt.

Binärbäume im Objekt-Algebra-Stil

Betrachten wir zuerst eine Implementation von Binärbäumen im Visitor-Stil (völlig analog zum Interpreter im Visitor-Stil – hier ein Vergleich des funktionalen Pattern-Matching-, Visitor und Objekt-Algebra-Stils):

```
sealed abstract class BTree
case class Node(l: BTree, r: BTree) extends BTree
case class Leaf(n: Int) extends BTree

case class Visitor[T](node: (T,T) => T, leaf: Int => T)

def foldExp[T](v: Visitor[T], b: BTree) : T = b match {
  case Node(l,r) => v.node(foldExp(v,l), foldExp(v,r))
  case Leaf(n)   => v.leaf(n)
}

val sumVisitor = new Visitor[Int]((l,r) => l+r, n => n)

assert(foldExp(sumVisitor, Node(Leaf(1),Leaf(2))) == 3)
```

Im Objekt-Algebra-Stil wird der Datentyp nicht durch eine abstrakte Klasse mit verschiedenen *Cases*, sondern durch die Funktionen der Faltungsoperation definiert, die Daten werden also durch ihre eigene Faltung repräsentiert:

```
trait BTreeInt[T] {
  def node(l: T, r: T) : T
  def leaf(n: Int) : T
}

def ex1[T](semantics: BTreeInt[T]) : T = {
  import semantics._
  node(leaf(1),leaf(2))
}

object TreeSum extends BTreeInt[Int] {
  def node(l: Int, r: Int) = l+r
  def leaf(n: Int) = n
}

assert(ex1(TreeSum) == 3)
```

Dadurch muss zur Durchführung einer Faltung nur eine Instanz von `BTreeInt` (hier etwa `TreeSum`) an einen Wert des Datentyps (hier etwa `ex1`) überreicht werden, in der die Faltungsoperation für beide Fälle konkret definiert sind.

Eigenschaften des Objekt-Algebra-Stils: - Jeder Konstruktor des Datentyps (hier `Node` und `Leaf`) wird zu einer Funktion in der abstrakten "Signaturklasse" mit Typparameter `T` (hier `BTreeInt[T]`), wobei rekursive Vorkommen des Datentyps durch `T` ersetzt werden.

- Konkrete Werte des Datentyps sind als Funktionen mit Typparameter `T` definiert, deren Eingabe eine Instanz der Signaturklasse und deren Ausgabe vom Typ `T` ist.
- Konkrete Faltungen werden zu Instanziierungen der Signaturklasse (hier `TreeSum`), in der die Faltungsfunktionen für die verschiedenen Fälle definiert werden.

Die Argumente der Faltungsfunktion werden also in einem `Trait` zusammengefasst und können als Objekt überreicht werden. Dadurch lässt sich der Datentyp durch Vererbung erweitern (etwa um Blätter, die Strings enthalten):

```
trait BTreeMixed[T] extends BTreeInt[T] {
  def leaf(s: String) : T
}

def ex2[T](semantics: BTreeMixed[T]) : T = {
  import semantics._
  node(leaf(1),node("a"))
}
```

Peano-Zahlen im Objekt-Algebra-Stil

Wir können die Church-Kodierungen für Zahlen folgendermaßen in Scala nachbilden, wobei Zahlen Instanzen von `Num` sind und (entsprechend der Church-Kodierung) aus ihrer eigenen Faltungsfunktion bestehen:

```
trait Num {
  def fold[T](s: T => T, z: T) : T
}

case object Zero extends Num {
  def fold[T](s: T => T, z: T) : T = z
}

case object One extends Num {
  def fold[T](s: T => T, z: T) : T = s(z)
}

def succ(n: Num) : Num = {
  new Num {
    def fold[T](s: T => T, z: T) : T = s(n.fold(s,z))
  }
}
```

Bei der Implementierung im Objekt-Algebra-Stil werden die Funktionen der Faltung (also Parameter von `fold`) wieder zu einem Objekt zusammengefasst:

```
trait NumSig[T] {
  def s(p: T) : T
  def z: T
}
```

```
trait Num { def apply[T](x: NumSig[T]) : T }
```

Eine Objekt-Algebra ist eine Klasse, die ein generisches *Abstract Factory Interface* implementiert, das im wissenschaftlichen Gebiet der *universellen Algebra* als *algebraische Signatur* bezeichnet wird. Die obige Implementation im Objekt-Algebra-Stil ist eine *algebraische Struktur* mit den Operationen `s` und `z`. Der Typ der Operationen (hier `NumSig`) wird als *Signatur* oder *Funktor* bezeichnet, bei `Num` handelt es sich um eine *(Funktor-)Algebra*.

Eine **algebraische Struktur** besteht gewöhnlich aus einer nichtleeren Menge (*Grundmenge* oder *Trägermenge*) und einer Familie von *inneren Verknüpfungen* (*Grundoperationen*) auf der Menge. Ein Beispiel für eine einfache algebraische Struktur ist etwa das *Monoid*.

Ein **Monoid** ist eine algebraische Struktur bestehend aus einer Menge M, einer Abbildung $\odot : M \times M \to M$ und einem neutralen Element $e \in M$, so dass $\forall a \in M$ gilt: $e \odot a = a \odot e = a$. Außerdem gilt $\forall a, b, c \in M$: $(a \odot b) \odot c = a \odot (b \odot c)$ (*Assoziativität*).

Wir implementieren Zahlen und Addition im Objekt-Algebra-Stil folgendermaßen:

```
val zero: Num = new Num { def apply[T](x: NumSig[T]): T = x.z }
val one: Num = new Num { def apply[T](x: NumSig[T]): T = x.s(x.z) }
val two: Num = new Num { def apply[T](x: NumSig[T]) : T = x.s(one.apply(x))}

def plus(a: Num, b: Num) : Num = new Num {
  def apply[T](x: NumSig[T]) : T = a.apply(new NumSig[T] {
    def s(p: T): T = x.s(p)
    def z: T = b.apply(x)
  })
}
```

Dieser Stil ermöglicht verschiedene konkrete Implementierungen des Interfaces, bei denen die Argumente für die Faltung übergeben werden, die die `Num`-Objekte in der `apply`-Funktion noch erwarten.

```
object NumAlg extends NumSig[Int] {
  def s(x: Int) : Int = x+1
  def z = 0
}

assert(two(NumAlg) == 2)
assert(plus(one,two)(NumAlg) == 3)
```

Wir können bspw. das Objekt `NumAlg` überreichen, um die Church-kodierten Zahlen in Scala-Integer umzuwandeln. Dabei wird implizit `apply` mit der übergebenen `NumSig`-Instanziierung aufgerufen.

In diesem Fall bietet der Objekt-Algebra-Stil keine speziellen Vorteile, anders ist es aber bei unserem Interpreter.

Interpreter im Objekt-Algebra-Stil

Der Objekt-Algebra-Stil erzwingt Kompositionalität, deshalb wählen wir zur Umwandlung unsere kompositionale Implementation von FAE, in der Closures durch Metainterpretation umgesetzt sind.

```
trait Exp[T] {
  implicit def num(n: Int) : T
  implicit def id(name: String) : T
  def add(l: T, r: T) : T
  def fun(param: String, body: T) : T
  def app(fun: T, arg: T) : T
  def wth(x: String, xDef: T, body: T) : T = app(fun(x,body),xDef)
```

```
}

sealed abstract class Value
type Env = Map[String,Value]
case class ClosureV(f: Value => Value) extends Value
case class NumV(n: Int) extends Value

trait eval extends Exp[Env => Value] {
  def id(x: String) : Env => Value = env => env(x)
  def fun(p: String, b: Env => Value) : Env => Value =
    env => ClosureV(v => b(env+(p -> v)))
  def app(fun: Env => Value, arg: Env => Value) : Env => Value =
    env => fun(env) match {
      case ClosureV(f) => f(arg(env))
      case _ => sys.error("Can only apply functions")
    }
  def num(n: Int) : Env => Value = _ => NumV(n)
  def add(l: Env => Value, r: Env => Value) : Env => Value =
    env => (l(env),r(env)) match {
      case (NumV(a),NumV(b)) => NumV(a+b)
      case _ => sys.error("Can only add numbers")
    }
}

object eval extends eval

def test[T](semantics: Exp[T]) = {
  import semantics._
  app(app(fun("x",fun("y",add("x","y"))),5),3)
}

assert(test(eval)(Map()) == NumV(8))
```

Dieser Stil erlaubt vollständige Modularität, so dass Sprachkonstrukte nach Bedarf hinzugefügt werden können. Wir können die Sprache etwa folgendermaßen um Multiplikation erweitern:

```
trait ExpWithMul[T] extends Exp[T] {
  def mul(l: T, r: T) : T
}

object evalWithMul extends eval with ExpWithMul[Env => Value] {
  def mul(l: Env => Value, r: Env => Value) : Env => Value =
    env => (l(env),r(env)) match {
      case (NumV(a),NumV(b)) => NumV(a*b)
      case _ => sys.error("Can only multiply numbers")
    }
}
def test2[T](semantics: ExpWithMul[T]) = {
  import semantics._
  app(app(fun("x",fun("y",mul("x","y"))),5),3)
}

assert(test2(evalWithMul)(Map()) == NumV(15))
```

Expression Problem

Bei unserem Interpreter gibt es zwei Arten von Erweiterung: Zum einen gibt es die Erweiterung um zusätzliche Funktionen neben `eval` (bspw. `print` oder `countNodes`), zum anderen gibt es die Erweiterung um zusätzliche Sprachkonstrukte. - Unsere bisherige (funktionale) Implementation mit Pattern Matching erlaubt die modulare Erweiterung um neue Funktionen – es können neben `eval` weitere strukturell rekursive Funktionen angelegt werden, die auf Expressions operieren. Die Erweiterung um neue Sprachkonstrukte ist aber nicht modular möglich, denn die Funktionen können nach ihrer Definition nicht mehr um zusätzliche Fälle erweitert werden.

```
sealed trait Exp
case class Num(n: Int) extends Exp
case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp) : Int = e match {
  case Num(n) => n
  case Add(l,r) => eval(l) + eval(r)
}

// ...
```

- Bei einer objektorientierten Implementierung können Funktionen nicht nachträglich hinzugefügt werden, da diese in jedem Konstruktor definiert werden müssen. Modulare Ergänzung neuer Sprachkonstrukte ist hingegen gut möglich, die abstrakte Oberklasse kann nachträglich erweitert werden.

```
sealed abstract class Exp {
  def eval() : Int
}

case class Num(n: Int) extends Exp {
  def eval() = n
}

case class Add(l: Exp, r: Exp) extends Exp {
  def eval() = l.eval + r.eval
}

// ...
```

Der große Vorteil des Objekt-Algebra-Stils ist die modulare Erweiterbarkeit in beiden Dimensionen. Es können sowohl neue Sprachkonstrukte modular ergänzt, als auch neue Funktionen neben `eval` hinzugefügt werden. Es handelt sich dabei um eine Lösung für das sogenannte *Expression Problem*.

Das **Expression Problem** beschreibt die Suche nach einer Datenabstraktion, bei der sowohl neue Datenvarianten (Cases), als auch neue Funktionen, die auf dem Datentyp operieren, ergänzt werden können, ohne dass bisheriger Code modifiziert werden muss und wobei Typsicherheit gewährt bleibt (Wikipedia-Artikel).

Bei einem funktionalen Ansatz ist typischerweise die Erweiterung mit Operationen gut unterstützt, bei einem objektorientierten Ansatz hingegen die Erweiterung mit neuen Datenvarianten.

Webprogrammierung mit Continuations

Angenommen, man will eine interaktive Webanwendung über mehrere Webseiten hinweg programmieren, so wird man vor eine Herausforderung gestellt: Das Webprotokoll HTTP ist zustandslos, d.h. ein Programm im Web terminiert nach jeder Anfrage. Anfragen sind unabhängig voneinander und es ist kein Zugriff auf vorherige Anfragen und dabei übermittelte Daten möglich. Betrachten wir etwa die interaktive Funktion `progSimple` im folgenden Code:

```
import scala.io.StdIn.readLine

def inputNumber(prompt: String) : Int = {
  println(prompt)
  Integer.parseInt(readLine())
}

def progSimple() : Unit = {
  println(inputNumber("First number:") + inputNumber("Second number:"))
}
```

Hier finden nacheinander zwei Eingaben durch den Nutzer statt, wobei das Ergebnis aus beiden Werten berechnet wird. Würde man diese Funktion im Web umsetzen wollen, so dass der Nutzer auf zwei verschiedenen Seiten die Werte eingibt und absendet und auf einer dritten Seite das Ergebnis angezeigt bekommt, dann wäre aufgrund der Zustandslosigkeit von HTTP ein spezieller Programmierstil notwendig.

Der Ablauf zerfällt dabei in die folgenden Teilprogramme: - **Teilprogramm** a zeigt das Formular für die erste Zahl an. - **Teilprogramm** b konsumiert die Zahl aus dem ersten Formular und generiert das Formular für die zweite Zahl. - **Teilprogramm** c konsumiert die Daten aus dem zweiten Formular, berechnet die Ausgabe und erzeugt die Seite mit dem Ergebnis.

Nun stellt sich die Frage, wie im zustandslosen Protokoll das Teilprogramm c auf die in a eingegebenen Daten zugreifen kann. Hierzu muss der eingegebene Wert über b weitergereicht werden, etwa als verstecktes Formularfeld in HTML oder als Parameter in der URL.

Wir können die Zustandslosigkeit mit dem Rückgabetyp `Nothing` modellieren, dabei brechen wir alle Funktionen mit einem Fehler ab, so dass sie nicht zurückkehren.

```
def webDisplay(s: String) : Nothing = {
  println(s)
  sys.error("Program terminated")
}

def webRead(prompt: String, continue: String) : Nothing = {
  println(prompt)
  println("Send input to "+continue)
  sys.error("Program terminated")
}

def progA = webRead("First number:", "progB")
def progB(n: Int) = webRead("First number was "+n+"\nSecond number:", "progC")
def progC(n1: Int, n2: Int) = webDisplay("Sum of "+n1+" and "+n2+" is "+(n1+n2))
```

Hier wird die Weitergabe der Daten von einem Teilprogramm zum nächsten nur durch die Ausgaben und Eingaben in der Konsole (also händisch durch den Nutzer) modelliert, d.h. es wird bspw. erst `progA` mit `2` aufgerufen, dann `progB` mit `3` und zuletzt `progC` mit `2` und `3`. Dadurch ist der Nutzen der durchgeführten Programmtransformation noch nicht sonderlich klar erkennbar.

Implementierung mit Continuations

Es können aber auch die jeweils noch notwendigen Schritte als *Continuation* repräsentiert werden, im Fall von `progA` muss etwa noch die zweite Zahl eingelesen werden, dann müssen beide Zahlen addiert und das Ergebnis ausgegeben werden. In der ursprünglichen Variante des Programms

```
def progSimple() : Unit = {
  println(inputNumber("First number:") + inputNumber("Second number:"))
}
```

entspricht das der folgenden anonymen Funktion:

```
val cont1 = (n: Int) => println(n + inputNumber("Second number:"))
```

Die Continuation zum Zeitpunkt der zweiten Eingabe entspricht dem folgenden Wert:

```
val cont2 = (m: Int) => println(n + m)
```

Durch das Ablegen dieser Continuations in einer Map und der Ausgabe des zugehörigen Schlüssels können wir `webRead` umgestalten und es dem Nutzer ermöglichen, die Auswertung ab dem letzten Zwischenstand fortzusetzen.

```
val continuations = new mutable.HashMap[String, Int=>Nothing]
var nextIndex : Int = 0
def getNextId : String = {
  nextIndex += 1
  "c"+nextIndex
}

def webRead_k(prompt: String, k: Int => Nothing) : Nothing = {
  val id = getNextId
  continuations += (id -> k)
  println(prompt)
  println("To continue, invoke continuation "+id)
  sys.error("Program terminated")
}

def continue(kId: String, input: Int): Nothing = continuations(kId)(input)

def webProg =
  webRead_k("First number:", (n: Int) =>
    webRead_k("Second number:", (m: Int) =>
      webDisplay("Sum of "+n+" and "+m+" is "+(n+m))))
```

Nun kann zuerst `webProg` aufgerufen werden, es wird die ID für die nächste Continuation ausgegeben. Mit dieser ID und der ersten Zahl kann dann `continue` aufgerufen werden, die dabei ausgegebene ID kann dann `continue` mit der zweiten Zahl übergeben werden, woraufhin das Ergebnis angezeigt wird. Die Bezeichner `n` und `m` sind zum Zeitpunkt, zu dem `webDisplay` aufgerufen wird, durch Scalas interne Closures gebunden und können korrekt aufgelöst werden.

In Bezug auf Webprogrammierung entsprechen die Continuations Zwischenzuständen der Auswertung, die serverseitig hinterlegt werden. Dabei erhält der Client (im Hintergrund) einen Bezeichner, um diesen Zustand mit der nächsten Eingabe aufzurufen. Somit könnte auch beim Klonen des Tabs oder bei Verwendung des "Zurück"-Buttons das Programm in allen Instanzen korrekt fortgesetzt werden. Das steht im Kontrast zu einer Implementierung mit einer *Session*, wobei der Client (und nicht der Zustand) anhand einer übermittelten Session-ID erkannt wird. In diesem Fall könnte es bei mehreren Instanzen zu fehlerhaften Ergebnissen kommen, da alle Instanzen über eine Session-ID laufen und damit nicht unabhängig sind.

Diese Eigenschaft ist auch bei unserer Implementierung erkennbar, es kann eine ausgegebene ID mehrfach aufgerufen werden, wobei die Auswertung jeweils dann mit der nächsten ID korrekt fortgesetzt werden kann und das korrekte Ergebnis liefert.

Eine Continuation kann als Repräsentation des Call-Stacks in einer Funktion aufgefasst werden.

Rekursion im Web-Stil

Betrachten wir nun den allgemeineren Fall der n-fachen Addition: Im folgenden Programm wird eine Liste von Gegenständen rekursiv durchlaufen, wobei der Nutzer für jeden Gegenstand aufgefordert wird, einen Preis einzugeben. Nachdem alle Listenelemente abgearbeitet wurden, wird die Summe der Zahlen ausgegeben.

```
def inputNumber(prompt: String) : Int = {
  println(prompt)
  Integer.parseInt(readLine())
}

def addAllCosts(il: List[String]): Int = il match {
  case List() => 0
  case first :: rest => inputNumber("Cost of "+first+":") + addAllCosts(rest)
}

val testList = List("Banana", "Apple", "Orange")
def test() : Unit = println("Total cost: " + addAllCosts(testList))
```

Dieses Programm besitzt nach der "Web-Stil"-Transformation die Form:

```
def addAllCosts_k(il: List[String], k: Int => Nothing) : Nothing = il match {
  case List() => k(0)
  case first :: rest =>
    webRead_k("Cost of "+first+":",
      (n: Int) => addAllCosts_k(rest, (m: Int) => k(m+n)))
}

def testWeb() : Unit = addAllCosts_k(testList, m => webDisplay("Total cost: "+m))
```

Die Funktion `addAllCosts_k` wird aufgerufen mit der Liste von Gegenständen und der Contination `m => webDisplay("Total cost: "+m)`. Im Fall der leeren Liste wird die Continuation `k` auf `0` aufgerufen, es würde also `Total cost: 0` angezeigt werden. Ist die Liste nicht leer, so wird durch `webRead_k` der Nutzer dazu aufgefordert, den Preis des ersten Gegenstandes einzugegeben. Dabei wird die Continuation `n => addAllCosts_k(rest, m => k(m+n))` übergeben, diese Continuation wird also vom Nutzer/Client als nächstes mit dem entsprechenden Preis aufgerufen. Dabei wird die Continuation `m => k(m+n)` an `addAllCosts_k` zurückgereicht, d.h. die Nutzereingabe `n` wird jeweils den Kosten hinzugefügt.

Die Funktion `addAllCosts` lässt sich geschickt mit `map` umformulieren:

```
def addAllCostsMap(items: List[String]) : Int = {
  items.map((s: String) => inputNumber("Cost of " + s + ":")).sum
}
```

Würden wir auf dieser Implementation die Web-Transformation anwenden wollen, so müssten wir auch `map` transformieren. Die Web-Transformation ist nämlich "allumfassend" und betrifft alle Funktionen, die in einem Programm auftreten (bis auf primitive Operationen). Es benötigen alle Funktionen einen Continuation-Parameter, damit sie auch innerhalb anderer Programme im Web-Stil eingesetzt werden können. Wir müssen in diesem Fall also `map` im Web-Stil verfassen. Oben ist die "normale" Implementation von `map`, unten die Web-transformierte Variante:

```
def map[X,Y](l: List[X], f: X => Y) : List[Y] = c match {
  case List() => List()
  case x::xs => f(x)::map(xs,f)
}

def map_k[X,Y](l: List[X],
               f: X, (Y => Nothing) => Nothing,
               k: List[Y] => Nothing) : Nothing = l match {
  case List() => k(List())
  case x::xs => f(x, y => map_k(xs, f, (ys: List[Y]) => k(y::ys)))
}
```

Bei der Transformation wird überall der Rückgabetyp mit `Nothing` ersetzt, die ursprünglichen Rückgabewerte

werden stattdessen an die Continuation gereicht. So wird `f` nun durch ein neues Argument mit dem Typ `Y => Nothing` ergänzt, der Typ der Eingabe entspricht dem ursprünglichen Rückgabetyp. Die `map`-Funktion selbst wird durch ein neues Argument mit dem Typ `List[T] => Nothing` ergänzt, auch hier entspricht der Typ der Eingabe dem ursprünglichen Rückgabetyp.

Die Auswertungsreihenfolge ist entscheidend für die Web-Transformation, die implizite Auswertungsreihenfolge des Ausgangsprogramms (etwa bei geschachtelten Ausdrücken) muss explizit ausformuliert werden, das Programm wird *sequentalisiert*.

Außerdem ist die Transformation global, es müssen alle verwendeten Funktionen (auch aus Bibliotheken etc.) transformiert werden.

Aufgrund des Aspekts der Sequentialisierung ist die Transformation mit Continuations auch für das Programmieren von Compilern relevant. Man spricht bei diesem "Web-Stil" auch von *Continuation Passing Style* (*CPS*).

Continuation Passing Style

Programme in CPS haben die folgenden Eigenschaften: - Alle Zwischenwerte besitzen einen Namen. - Die Auswertungsschritte werden sequentialisiert (und die Auswertungsreihenfolge ist somit explizit). - Alle Ausdrücke erhalten einen Continuation-Parameter und liefern keinen Rückgabewert (Rückgabetyp `Nothing`), sondern rufen den Continuation-Parameter auf ihrem Ergebnis auf. - Alle Funktionsaufrufe sind *Tail Calls* (da Funktionen nicht zurückkehren).

Man spricht von einem **Tail Call**, wenn bei einem rekursiven Aufruf im Rumpf einer Funktion keine weitere Berechnung nach der Rückkehr des Aufrufs stattfindet. Der Aufruf `f(n+1)` in `def f(n: Int): Int = f(n+1)` ist ein Tail Call, in `def f(n: Int): Int = f(n+1)*2` jedoch nicht.

Liegt nach der CPS-Transformation eine "triviale" Continuation (d.h. `k` bleibt unverändert) bei einem rekursiven Aufruf vor, so lag ursprünglich ein Tail Call vor. Das wird bspw. an der folgenden Funktion deutlich:

```
def sumAcc(n: Int, acc: Int) : Int = l match {
  case 0 => acc
  case n => sumAcc(n-1, n+acc)
}

def sumAcc_k(n: Int, acc: Int, k: Int => Nothing) : Nothing = l match {
  case 0 => k(acc)
  case n => sumAcc_k(n-1, n+acc, k)
}
```

Die Funktion berechnet die Summe aller Zahlen von 1 bis `n`, verwendet aber im Gegensatz zu einer herkömmlichen Lösung (`case n => n+sum(n-1)`, siehe `sum` im Beispiel weiter unten) einen zusätzlichen Parameter, in dem die aktuelle Zwischensumme gehalten wird. Dadurch liegt im rekursiven Fall ein Tail Call vor und in der CPS-transformierten Variante wird `k` unverändert weitergereicht, was bei der herkömmlichen Lösung nicht der Fall wäre.

Bei Rekursion mit Tail Call (*Endrekursion*) muss der Kontext des rekursiven Aufrufs nicht auf dem Call Stack gespeichert werden, in Scala wird deshalb einfache Endrekursion erkannt und entsprechend optimiert. In Java werden auch endrekursive Aufrufe auf dem Stack hinterlegt, wodurch rekursive Berechnungen immer einen größeren Speicherverbrauch haben als iterative. In Racket findet bei Endrekursion nie eine "Kontextanhäufung" auf dem Call-Stack statt.

Bei der CPS-Transformation von Funktionen und Werten sind die folgenden Schritte notwendig: 1. Ersetzen aller Rückgabetypen durch `Nothing`, wobei auch der Typ bei Konstanten `c: T` durch `T => Nothing` ersetzt wird

2. Ergänzen eines Parameters `k` mit Typ `R => Nothing`, wobei `R` der ursprüngliche Rückgabetyp ist (Konstanten der Form `c: T` werden also in Funktionen der Form `c_k(k: T => Nothing): Nothing` umgewandelt)
3. Weitergabe des Ergebnisses an `k`, bei Konstante `val c: T = x` etwa `def c_k(k: T => Nothing): Nothing = k(x)`
4. Sequentialisierung durch Weiterreichen der Zwischenergebnisse, aus `f(f(42))` wird bspw. `k => f_k(42, fRes => f_k(fRes,k))` oder aus `f(1) + g(2)` wird `k => f_k(1, fRes => g_k(2, gRes => k(fRes+gRes)))`

Weitere Beispiele der CPS-Transformation:

```
// constant value
val x: Int = 42
def x_k(k: Int => Nothing) : Nothing = k(42)

// recursion with "context"
def sum(n: Int) : Int = n match {
  case 0 => 0
  case n => n + sum(n-1)
}
def sum_k(n: Int, k: Int => Nothing) : Nothing = n match {
  case 0 => k(0) // k called with result
  case n => sum_k(n-1, m => k(n+m)) // non-trivial continuation
}

// two-way tail call recursion
def even(n: Int) : Boolean = n match {
  case 0 => true
  case n => odd(n-1)
}
def odd(n: Int) : Boolean = n match {
  case 0 => false
  case n => even(n-1)
}

def even_k(n: Int, k: Boolean => Nothing) : Nothing = n match {
  case 0 => k(true)
  case n => odd_k(n-1,k) // trivial continuation
}
def odd_k(n: Int, k: Boolean => Nothing) : Nothing = n match {
  case 0 => k(false)
  case n => even_k(n-1,k) // trivial continuation
}
```

Automatische CPS-Transformation

Nun wollen wir die CPS-Transformation von Ausdrücken in FAE automatisieren. Die Transformationsregeln können folgendermaßen formalisiert werden:

- **Konstanten:** Aus `c` wird `k => k(c)`
- **Funktionsdefinitionen:** Aus `x => y` wird `k => k( (x, dynK) => dynK(y) )`
- **Funktionsapplikationen:** Aus `f(x)` mit `f: X => Y` wird `(k: Y => ...) => f_k(x, y => k(y))`

Im Fall von Funktionsdefinition müssen zwei Continuations beachtet werden: Die Continuation vom Zeitpunkt der Definition (`k`) sowie die *dynamische Continuation* (`dynK`), die bei der Funktionsapplikation überreicht wird.

Wir definieren einen zweiten Typ neben `Exp` um CPS-transformierte Ausdrücke zu repräsentieren, da sich zum einen die Syntax mancher Sprachkonstrukte durch die Transformation ändert (es kommt der zusätzliche Continuation-Parameter hinzu) und zum anderen um die Eigenschaften von CPS-transformierten Programmen explizit zu formulieren (und sicherzustellen).

```
sealed abstract class CPSExp
sealed abstract class CPSVal extends CPSExp
case class CPSNum(n: Int) extends CPSVal
case class CPSFun(x: String, k: String, body: CPSExp) extends CPSVal
case class CPSCont(v: String, body: CPSExp) extends CPSVal

case class CPSVar(x: String) extends CPSVal { override def toString: String = x }
implicit def string2cpsExp(s: String): CPSVar = CPSVar(s)

case class CPSAdd(l: CPSVar, r: CPSVar) extends CPSVal
case class CPSFunApp(f: CPSVar, a: CPSVar, k: CPSVar) extends CPSExp
case class CPSContApp(k: CPSVal, a: CPSVal) extends CPSExp
```

Wir unterscheiden zwei syntaktische Kategorien, nämlich `CPSVal` (Werte) und `CPSExp` (Ausdrücke, die keinen Rückgabewert besitzen). Addition betrachten wir als primitive Operation, die nicht CPS-transformiert wird, und zählen diese somit zu `CPSVal`. Zu den Werten gehört auch die Repräsentation von Bezeichnern, `CPSVar`. Es wird zwischen Funktionsdefinitionen und Continuations unterschieden, wodurch auch die Applikation von Funktionen und Continuations getrennte Sprachkonstrukte sind. Bei Funktions- und Continuationapplikationen haben die Argumente den Typ `CPSVar`, somit kann es sich nicht um geschachtelte Ausdrücke handeln, diese würden nämlich nicht CPS entsprechen.

Wir benötigen wieder einen Mechanismus, um "frische" Bezeichner zu generieren, diesen übernehmen von unserem substitutionsbasierten FAE-Interpreter:

```
def freeVars(e: Exp) : Set[String] =  e match {
   case Id(x) => Set(x)
   case Add(l,r) => freeVars(l) ++ freeVars(r)
   case Fun(x,body) => freeVars(body) - x
   case App(f,a) => freeVars(f) ++ freeVars(a)
   case Num(n) => Set.empty
}
def freshName(names: Set[String], default: String) : String = {
  var last : Int = 0
  var freshName = default
  while (names contains freshName) { freshName = default+last; last += 1; }
  freshName
}
```

Die CPS-Transformation von FAE-Ausdrücken läuft folgendermaßen ab:

```
def cps(e: Exp) : CPSCont = e match {
  case Num(n) => {
    CPSCont("k", CPSContApp("k", CPSNum(n)))
  }
  case Id(x) => {
    val k = freshName(freeVars(e),"k")
    CPSCont(k, CPSContApp(k, CPSVar(x)))
```

```
  }
  case Add(l,r) => {
    val k = freshName(freeVars(e),"k")
    val lv = freshName(freeVars(r),"lv")
    CPSCont(k, CPSContApp(cps(l), CPSCont(lv,
      CPSContApp(cps(r), CPSCont("rv",
        CPSContApp(k, CPSAdd(lv,"rv")))))))
  }
  case Fun(p,b) => {
    val k = freshName(freeVars(e),"k")
    val dynK = freshName(freeVars(e),"dynK")
    CPSCont(k, CPSContApp(k, CPSFun(p, dynK, CPSContApp(cps(b), dynK))))
  }
  case App(f,a) => {
    val k = freshName(freeVars(e),"k")
    val fv = freshName(freeVars(a),"fv")
    CPSCont(k, CPSContApp(cps(f), CPSCont(fv,
      CPSContApp(cps(a), CPSCont("av",
        CPSFunApp(fv,"av",k))))))
  }
}
```

Entsprechend der zu Beginn formulierten Regeln zur Transformation werden Konstanten (`Num`- und `Id`-Ausdrücke) umgewandelt von `c` in `k => k(c)`, was als `CPSExp` dem Ausdruck `CPSCont(k, CPSContApp(k, c))` entspricht. Im `Add`- und `App`-Fall werden die zwei Unterausdrücke sequentiell umgewandelt, wobei die Zwischenergebnisse jeweils an einen Bezeichner (`CPSVar`, implizite Umwandlung von Strings) gebunden werden.

Der Bezeichner `k` darf jeweils nicht in `e` vorkommen, der Bezeichner für den transformierten linken Unterausdruck (`lv` bzw. `fv`) darf nur nicht im rechten Unterausdruck vorkommen und der Bezeichner für den transformierten rechten Unterausdruck kann frei gewählt werden, da zwischen dem bindenden Vorkommen und der Verwendung kein rekursive Transformation eines Unterausdrucks stattfindet.

Im `Fun`-Fall wird der Rumpf mit der dynamischen Continuation transformiert und wird auch in den `CPSFun`-Ausdruck eingefügt. Auf diesen wird dann die Continuation vom Zeitpunkt der Definition angewendet.

Bei der hier verwendeten Transformation handelt es sich um die sogenannte *Fischer-Transformation*.

Die **Fischer-CPS-Transformation** ist ein möglicher CPS-Transformationsalgorithmus von vielen verschiedenen. Der Vorteil der Algorithmus ist seine Einfachheit und die Tatsache, dass es sich um eine strukturelle Rekursion des abstrakten Syntaxbaums handelt. Ein großer Nachteil ist aber, dass sogenannte *administrativen Redexe* bei der Umwandlung entstehen, dabei handelt es sich um Contination-Applikationen von anonymen Funktionen, die nicht im ursprünglichen Programm enthalten waren und direkt aufgelöst werden könnten.

Bspw. ergibt die Fischer-Transformation von `Add(2,3)` den Ausdruck

```
CPSCont("k", CPSContApp(CPSCont("k", CPSContApp("k",2)),
  CPSCont("lv", CPSContApp(CPSCont("k", CPSContApp("k",3)),
    CPSCont("rv", CPSAdd("rv","lv"))))))
```

anstelle von

```
CPSCont("k", CPSContApp("k", CPSAdd(2,3)))
```

Fortgeschrittenere Transformationsalgorithmen versuchen möglichst viele dieser administrativen Redexe zu vermeiden.

First-Class Continuations

In Programmiersprachen mit *First-Class Continuations* (bspw. Scheme, Racket) gibt es Sprachkonstrukte, um die aktuelle Continuation zu jedem Zeitpunkt abzugreifen und um damit zu arbeiten (also um die Continuation zu reifizieren, zu binden, als Parameter zu übergeben oder aufzurufen). Mit solch einem Sprachfeature kann der Programmierer bspw. fortgeschrittene Kontrollstrukturen selbst definieren.

In Racket gibt es die Funktion `let/cc`, mit der die aktuelle Continuation an einen Identifier gebunden und im Rumpf von `let/cc` aufgerufen werden kann.

```
[>] (number->string (+ 1 (let/cc k (string-length (k 3)))))
"4"
```

Im obigen Beispiel werden die Funktionsaufrufe vor dem Aufruf von `let/cc` als Continuation an `k` gebunden, im Rumpf von `let/cc` wird dann `k` mit `3` aufgerufen und damit die Continuation fortgesetzt, es werden die in der Contination gespeicherten Funktionsaufrufe angewendet und `"4"` ausgegeben. Der Aufruf von `k` kehrt nicht zurück, wodurch der Funktionsaufruf von `string-length` zwischen `let/cc` und `k` nicht mehr auf das Ergebnis angewendet wird.

Die Continuation kann auch durch `set!` an einen globalen Identifier gebunden werden, um sie außerhalb des Rumpfes von `let/cc` aufrufen zu können:

```
[>] (define c "dummy")
[>] (number->string (+ 1 (let/cc k (begin (set! c k) (k 3)))))
"4"
[>] (c 5)
"6"
```

FAE mit First-Class-Continuations

Nun wollen wir unserem FAE-Interpreter First-Class-Continuations als Sprachfeature hinzufügen. Wir ergänzen dazu das folgende Sprachkonstrukt:

```
case class LetCC(param: String, body: Exp) extends Exp
```

Bei einem Aufruf von `LetCC` soll, wie in Racket, die aktuelle Contination an den Bezeichner `param` gebunden werden, wobei die Bindung im Rumpf von `LetCC` gültig ist.

In einem Programm, das bereits in CPS vorliegt, wäre das Bestimmen der aktuellen Continuation trivial. Für unsere Implementation von `LetCC` wollen wir aber nicht alle Programme transformieren, sondern stattdessen den Interpreter selbst in CPS verfassen. Continuations auf Interpreter-Ebene repräsentieren nämlich auch die noch durchzuführende Auswertung auf Ebene der Objektsprache.

Bei einer Implementierung durch die automatische CPS-Transformation von Programmen müsste jedes Programm erst transformiert werden. Der Interpreter muss hingegen nur ein Mal transformiert werden. Der erste Schritt zur Implementierung von `LetCC` ist also das Transformieren des Interpreters.

CPS-Transformation des Interpreters

Wir beginnen mit unserem FAE-Interpreter:

```
def eval(e: Exp, env: Env) : Value = e match {
  case Num(n) => NumV(n)
  case Id(x) => env(x)
  case Add(l,r) => (eval(l,env),eval(r,env)) match {
    case (NumV(a),NumV(b)) => NumV(a+b)
    case _ => sys.error("Can only add numbers")
  }
  case f@Fun(_,_) => ClosureV(f,env)
```

```
  case App(f,a) => eval(f,env) match {
    case ClosureV(f,cEnv) =>
      eval(f.body, cEnv+(f.param -> eval(a,env)))
    case _ => sys.error("Can only apply functions")
  }
}
```

Wir ersetzen den Rückgabetypen mit `Nothing` und ergänzen einen Continuation-Parameter `k` mit dem Typ `Value => Nothing`. Der `Num`-, `Id`- und `Fun`-Fall sind trivial, da diese nicht rekursiv sind, hier reichen wir das Ergebnis einfach an `k` weiter. Im `Add`- und `App`-Fall muss die Auswertung des linken und rechten Unterausdrucks sequentialisiert werden, wir werten von links nach rechts aus.

```
def eval(e: Exp, env: Env, k: Value => Nothing) : Nothing = e match {
  case Num(n) => k(NumV(n))
  case Id(x) => k(env(x))
  case Add(l,r) => eval(l, env, lv => eval(r, env, rv => (lv,rv) match {
    case (NumV(a),NumV(b)) => k(NumV(a+b))
    case _ => sys.error("Can only add numbers")
  }))
  case f@Fun(_,_) => k(ClosureV(f,env))
  case App(f,a) => eval(f, env, fv => fv match {
    case ClosureV(Fun(p,b),cEnv) =>
      eval(a, env, av => eval(b, cEnv+(p -> av), k))
    case _ => sys.error("Can only apply functions")
  })
}
```

Im `Add`-Fall werten wir erst den linken Unterausdruck aus und übergeben dabei eine Continuation, die das Ergebnis an `lv` bindet und mit der Auswertung des rechten Unterausdrucks fortfährt. Deren Ergebnis wird durch die übergebene Continuation an `rv` gebunden. Wie im ursprünglichen Interpreter werden nun durch Pattern Matching die Zahlen extrahiert und ein `NumV`-Objekt mit deren Summe erzeugt, dieses wird nun aber an `k` überreicht.

Im `App`-Fall wird auch erst der linke Teilausdruck ausgewertet und diesmal durch die Continuation an `fv` gebunden, durch Pattern Matching werden Parameter und Rumpf der Funktion sowie die Umgebung im Closure gebunden. Nun wird das Argument ausgewertet, durch die übergebene Continuation an `av` gebunden und zuletzt `eval` mit dem Rumpf, der Umgebung inkl. neuer Bindung und der Continuation `k` aufgerufen.

Um den Interpreter auch ohne Rückgabe testen zu können, verwenden wir folgende Funktion:

```
def startEval(e: Exp) : Value = {
  var res: Value = null
  val s: Value => Nothing = v => { res = v; sys.error("Program terminated") }
  try { eval(e, Map(), s) } catch { case _: Throwable => () }
  res
}
```

Es wird `eval` eine Continuation überreicht, die das Ergebnis der Auswertung bindet und dann mit einem Fehler die Auswertung beendet (um den Rückgabetyp `Nothing` zu erfüllen). Die Auswertung selbst wird in einem Try-Catch-Block gestartet, um den Fehler abzufangen. Das durch die Continuation gebundene Ergebnis wird von `startEval` ausgegeben.

Die CPS-Transformation des Interpreters hat zur Folge, dass der Interpreter nicht mehr vom Call-Stack der Hostsprache abhängig ist, da nun jeder Funktionsaufruf ein Tail Call ist und die noch notwendigen Auswertungsschritte jeweils in Form der Continuation überreicht werden.

Implementierung von 'LetCC'

Jetzt wo der Interpreter selbst CPS-transformiert ist, können wir das neue Sprachkonstrukt `LetCC` mit wenig Aufwand ergänzen, da wir die Continuations auf Interpreter-Ebene als Continuations der Objektsprache nutzen können. Zuerst erweitern wir `Exp` um das Sprachkonstrukt `LetCC`:

```
case class LetCC(param: String, body: Exp) extends Exp
```

Im Interpreter ergänzen wir den `LetCC`-Fall, hier setzen wir die Auswertung rekursiv im Body fort, wobei wir den Parameter in `LetCC` an die aktuelle Continuation `k` binden. Da `k` den Typ `Value => Nothing`, die Umgebung aber den Typ `Map[String,Value]` besitzt, können wir die Continuation nicht direkt in der Umgebung binden. Stattdessen erweitern wir `Value` um den Fall `ContV`:

```
case class ContV(k: Value => Nothing) extends Value
```

Nun sind Continations eine Werte-Art und können wie andere Werte an Identifier gebunden werden, wordurch wir den `LetCC`-Fall verfassen können:

```
case LetCC(p,b) => eval(b, env+(p -> ContV(k)))
```

Es fehlt noch die Applikation von Continuations, hierzu überladen wir das `App`-Konstrukt, so dass damit sowohl Funktionen als auch Continuations angewendet werden können. Im `App`-Fall müssen wir nun die Fallunterscheidung um einen `ContV`-Fall neben dem `ClosureV`-Fall erweitern:

```
case App(f,a) => eval(f, env, fv => fv match {
  case ClosureV(Fun(p,b),cEnv) =>
    eval(a, env, av => eval(b, cEnv+(p -> av), k))
  case ContV(k2) => eval(a, env, av => k2(av))
  case _ => sys.error("Can only apply functions")
})
```

Im `ContV`-Zweig werten wir das Argument aus und übergeben dabei eine Continuation, die das Ergebnis an `av` bindet und dann der aufzurufenden Continuation `k2` aus dem `ContV`-Objekt `av` überreicht. Dabei wird die aktuelle Continuation `k` ignoriert, die Auswertung "springt" ohne zurückzukehren. Der Wechsel zur Continuation `k2` kann als ein Austauschen des Call-Stacks aufgefasst werden.

Durch das "Wrappen" der Interpreter-Continuations (auf Ebene der Metasprache) können wir diese also als Continuations für die Objektsprache verwenden.

Delimited Continuations

Die (*Undelimited*) Continuations, die wir bisher kennen gelernt haben, führen bei einem Aufruf zu einem "Sprung" (ähnlich zu einer `GOTO`-Anweisung), wobei die Auswertung nicht mehr zur Stelle des Aufrufs zurückkehrt. Somit ist keine Komposition (d.h. "Nacheinanderschalten" und damit Kombinieren) von Continuations möglich.

Dies ist mit *Delimited Continuation* möglich, sie repräsentieren nur einen Ausschnitt des Call-Stacks (während *Undelimited Continuations* den gesamten Call-Stack repräsentieren) und kehren wie gewöhnliche Funktionen zurück. Dadurch ist die Komposition von Delimited Continuations möglich, sie werden deshalb auch als *Composable Continuations* bezeichnet.

Delimited Continuations sind ein sehr mächtiges Sprachkonstrukt und erlauben bspw. die Programmierung von fortgeschrittenem Exception Handling oder Backtracking-Algorithmen.

In der Racket-Bibliothek `racket/control` gibt es die zwei Funktionen `shift` und `reset`, mit denen Delimited Continuations erzeugt werden können. Dabei verhält sich `shift` ähnlich wie `let/cc`, wobei aber nur die Continuation ab dem (im AST) nächstliegenden Aufruf von `reset` gebunden wird, die Continuation wird also durch `reset` begrenzt (*delimitiert*) und repräsentiert nur den Ausschnitt des Call-Stacks zwischen `reset` und `shift`.

```
(* 2 (reset (+ 1 (shift k (k 5)))))
```

Im obigen Beispiel wird die Continuation, die durch `shift` an `k` gebunden wird, durch `reset` auf den Aufruf von `+` mit `1` beschränkt. Wird `k` auf `5` aufgerufen, so wird nur die Addition auf `5` durchgeführt, die Multiplikation vor `reset` gehört nicht zur Continuation. Somit entspricht die Berechnung `(* 2 (+ 1 5))` und das Ergebnis ist `12`. Da es sich bei `k` um eine Delimited Continuation handelt, ist auch Continuation-Komposition möglich:

```
(* 2 (reset (+ 1 (shift k (k (k 5))))))
```

In diesem Fall wird `k` zwei Mal auf `5` angewendet, die Berechnung entspricht also `(* 2 (+ 1 (+ 1 5)))` und das Ergebnis ist `14`.

Interpreter mit Delimited Continuations

Wir können im CPS-transformierten Interpreter mit wenig Aufwand Sprachkonstrukte hinzufügen, die `shift` und `reset` aus Racket entsprechen. Wir ergänzen dazu `Exp` um die zwei neuen Cases `Shift` und `Reset`:

```
case class Shift(param: String, body: Exp) extends Exp
case class Reset(body: Exp) extends Exp
```

Wir müssen den Typ des Interpreters ändern, da Aufrufe von Delimited Continuations zurückkehren und die Berechnung nicht abbrechen. Dadurch haben Continuations den Typ `Value => Value` und der Interpreter gibt einen `Value` zurück. Im Interpreter ergänzen wir die entsprechenden Fälle und passen den `App`-Fall an (Änderungen mit `<--` markiert):

```
def eval(e: Exp, env: Env, k: Value => Value) : Value = e match {
  // ...
  case App(f,a) => eval(f, env, fv => fv match {
    case ClosureV(Fun(p,b),cEnv) =>
      eval(a, env, av => eval(b, cEnv+(p -> av), k))
    case ContV(k2) => eval(a, env, av => k(k2(av))) // <--
    case _ => sys.error("Can only apply functions")
  })
  case Reset(e) => k(eval(e,env,x=>x)) // <--
  case Shift(p,b) => eval(b, env+(p -> ContV(k)), x=>x) // <--
}
```

Im `Reset`-Fall wird die aktuelle Continuation zurückgesetzt, indem die Identitätsfunktion anstelle von `k` weitergereicht wird. Im `Shift`-Fall wird wie im `LetCC`-Fall die aktuelle Continuation "umhüllt" und in der Umgebung an den Identifier gebunden. Außerdem wird hier auch die Continuation zurückgesetzt (weil dies dem Verhalten in Racket entspricht).

Im `App`-Fall wird bei der Applikation von Continuations die aktuelle Continuation nun nicht mehr verworfen, sondern wird mit dem Ergebnis der Continuation-Applikation aufgerufen, d.h. es findet eine Komposition der aktuellen Continuation nach der aufgerufenen Continuation statt.

Monaden

In unseren bisherigen Interpretern haben wir bereits einige verschiedene "Rekursions-Patterns" gesehen: - Im ersten Interpreter haben wir direkte Rekursion, bei der die `eval`-Funktion rekursiv auf den Unterausdrücken aufgerufen wird. Die rekursive Auswertung entspricht exakt der rekursiven Datenstruktur, in denen die Programme repräsentiert sind (*strukturelle Rekursion*).

- Bei der Einführung von Environments in AEId oder in FAE wird die Environment bei der rekursiven Auswertung im abstrakten Syntaxbaum (AST) nach unten weitergereicht, d.h. der zusätzliche Parameter `env` wird entlang der Datenstruktur propagiert.

- Bei der Ergänzung von mutierbaren Boxen in BCFAE haben wir den `Store`-Parameter hinzugefügt, der immer von der aktuellen Auswertungsposition zur nächsten Auswertungsposition gereicht (und dazwischen potentiell modifizert) wird.
- Der CPS-transformierte Interpreter entspricht dem Continuation Passing Style, den wir bereits ausführlich besprochen haben.

Monaden sind eine Möglichkeit, über solche (und noch viel mehr) Funktionskompositions-Patterns zu abstrahieren. Durch Monaden ist es möglich, einmalig verfassten Code in all diese Stile zu übersetzen.

Zur Einführung von Monaden wollen wir aber vorerst einen anderen Stil betrachten.

Einführung mit Option-Monade

```
def expr = h(!g(f(27)+"z"))
```

Angenommen, die Funktionen `f` und `h`, die im obigen Ausdruck aufgerufen werden, können in manchen Fällen keine Ausgabe liefern (was bspw. daran liegen könnte, dass sie intern Anfragen über ein Netzwerk schicken). In solch einem Fall wäre es sinnvoll, den `Option`-Datentyp zu verwenden, um auch bei fehlgeschlagener Auswertung `None` ausgeben zu können. Bei erfolgreicher Auswertung wird das Ergebnis mit `Some()` umhüllt:

```
def f(n: Int) : Option[String] = if (n < 100) Some("x") else None
def g(x: String) : Option[Boolean] = Some(x == "x")
def h(b: Boolean) : Option[Int] = if (b) Some(27) else None
```

Da aber Aufrufe der Funktionen potentiell `None` anstelle eines Ergebnisses liefern, muss `expr` modifiziert werden:

```
def expr = f(27) match {
  case Some(x) => g(x+"z") match {
    case Some(y) => h(!y)
        case None => None
  }
    case None => None
}
```

Bei jedem Aufruf muss anschließend ein Pattern-Match genutzt werden, um die zwei Fälle (`None` und `Some()`) zu unterscheiden und im `Some()`-Fall die Auswertung mit dem Ergebnis fortzusetzen. Es ist erkennbar, dass dieses Pattern bei jedem Aufruf einer Funktion mit Rückgabetyp `Option[T]` auftritt, wir abstrahieren also über das Pattern mit der folgenden Funktion:

```
def bindOption[A,B](a: Option[A], f: A => Option[B]) : Option[B] = a match {
  case Some(x) => f(x)
  case None => None
}

def expr =
  bindOption(f(27), (x: String) =>
    bindOption(g(x+"z"), (y: Boolean) =>
      h(!y)))
```

Mit `bindOption` können wir jeden Ausdruck im "Option-Stil" vereinfachen und die Redundanz des wiederholten, gleichartigen Pattern-Matchings vermeiden.

Angenommen, auf das verneinte Ergebnis von `g` wird nicht mehr `h` angewendet:

```
def expr2 = !g(f(27)+"z")
```

Dann muss das Ergebnis vor der Ausgabe wieder mit `Some()` "verpackt" werden, damit der Rückgabetyp `Option` (und damit der Option-Stil) erfüllt bleibt. Dies ist aber nicht mit `bindOption` möglich, sondern wir müssten `Some()` explizit aufrufen:

```
def expr2 =
  bindOption(f(27), (x: String) =>
    bindOption(g(x+"z"), (y: Boolean) =>
      Some(!y)))
```

Da wir aber über das Pattern der Funktionskomposition abstrahieren wollen, soll der `Option`-Datentyp nicht sichtbar sein. Wir fügen also unserem Funktionskompositions-Interface stattdessen neben `bindOption` eine zweite Funktion `unitOption` hinzu:

```
def bindOption[A,B](a: Option[A], f: A => Option[B]) : Option[B] = a match {
  case Some(x) => f(x)
  case None => None
}

def unitOption[A](a: A) : Option[A] = Some(a)
```

Mit `unitOption` können wir `expr2` folgendermaßen ausdrücken:

```
def expr2 =
  bindOption(f(27), (x: String) =>
    bindOption(g(x+"z"), (y: Boolean) =>
      unitOption(!y)))
```

Nun haben wir den Option-Stil abstrahiert, wir können aber einen Schritt weiter gehen und über den Typ `Option` abstrahieren, um beliebige Patterns auszudrücken. Dabei erhalten wir das *Monad-Interface*.

Definition

```
trait Monad[M[_]] {
  def unit[A](a: A) : M[A]
  def bind[A,B](m: M[A], f: A => M[B]) : M[B]
}
```

Eine Monade ist ein Tripel aus einem Typkonstruktor (`M[_]`) und zwei Funktionen, nämlich `unit` und `bind`. Es müssen zudem die folgenden *Monadengesetze* gelten: - `bind(unit(x),f) == f(x)` - `bind(x, y => unit(y)) == x` - `bind(bind(x,f),g) == bind(x, y => bind(f(y),g))`

D.h. `unit` ist eine Art "neutrales Element" und `bind` ist eine assoziative Operation.

Monaden dienen zur Funktionskomposition für Fälle, in denen der Rückgabetyp einer Funktion nicht dem Parametertyp der danach anzuwendenden Funktion entspricht, sondern ein "Zwischenschritt" notwendig ist.

Mit dem Monad-Interface kann das Einführungsbeispiel folgendermaßen ausgedrückt werden:

```
object OptionMonad extends Monad[Option] {
  override def unit[A](a: A) : Option[A] = Some(a)
  override def bind[A,B](m: Option[A], f: A => Option[B]) : Option[B] = m match {
    case Some(y) => f(y)
    case None => None
  }
}

def expr2(m: Monad[Option]) =
  m.bind(f(27), (x: String) =>
    m.bind(g(x+"z"), (y: Boolean) =>
      m.unit(!y)))

val expr2Res = expr2(OptionMonad)
```

For-Comprehension-Syntax

Die geschachtelten Aufrufe von `bind` sind bei komplexeren Beispielen etwas unleserlich, es kann aber die sogenannte *Monad-Comprehension*-Notation verwendet werden, um die Ausdrücke einfacher auszudrücken. Monad Comprehensions werden in Haskell und manchen anderen Sprachen nativ unterstützt, in Scala müssen wir stattdessen die *For-Comprehension*-Syntax für diese Zwecke "hijacken".

Diese Syntax wird im Normalfall für Listen und andere `Collection`-Datentypen verwendet:

```
val l = List(List(1,2),List(3,4))

val res = for {
  x <- l;
  y <- x } yield y+1 // == List(2,3,4,5)
```

Durch Desugaring werden For-Comprehensions in Aufrufe von `flatMap` und `map` umgewandelt:

```
val res = l.flatMap(x => x.map(y => y+1))
```

Diese Syntax kann für alle Datentypen angewendet werden, für die `map` und `flatMap` definiert ist. Mit der folgenden Funktionsdefinition kann somit die Syntax der For-Comprehensions für das Programmieren mit Monaden genutzt werden:

```
implicit def monadicSyntax[A, M[_]](m: M[A])(implicit mm: Monad[M]) = new {
  def map[B](f: A => B): Any = mm.bind(m, (x: A) => mm.unit(f(x)))
  def flatMap[B](f: A => M[B]): M[B] = mm.bind(m, f)
}
```

Durch die obige implizite Definition, in der wir `map` und `flatMap` so definieren, dass `map` der Kombination von `bind` und `unit` entspricht und `flatMap` der `bind`-Operation entspricht, sorgen wir dafür, dass die For-Comprehension-Syntax für die `Monad`-Klasse genutzt werden kann.

Unser Option-Monad-Beispiel kann mit dieser Syntax wie folgt ausgedrückt werden:

```
def expr2(m: Monad[Option]) = for {
  x <- f(27);
  y <- g(x+"z")
} yield !y
```

Operationen auf Monaden

Es lassen sich einige nützliche Operationen generisch für beliebige Monaden definieren:

`fmap` wandelt jede Funktion mit Typ `A => B` in eine Funktion vom Typ `M[A] => M[B]` um.

```
def fmap[M[_],A,B](f: A => B)(implicit m: Monad[M]): M[A] => M[B] =
  a => m.bind(a, (x: A) => m.unit(f(x)))

assert( fmap((n: Int) => n.toString)(OptionMonad)(Some(1)) == Some("1") )
```

`sequence` verknüpft eine Liste monadischer Werte zu einem einzelnen monadischen Wert, der eine Liste ist, aus einem Wert des Typs `List[M[A]]` wird ein Wert des Typs `M[List[A]]`.

```
def sequence[M[_],A](l: List[M[A]])(implicit m: Monad[M]) : M[List[A]] = l match {
  case x :: xs =>
      m.bind(x, (y: A) =>
        m.bind(sequence(xs), (ys : List[A]) =>
          m.unit(y :: ys)))
  case Nil => m.unit(List.empty)
}
```

```
def ex(implicit m: Monad[Option]) =
  List(m.unit(1),m.unit(2),m.unit(3))

// List[Option[Int]] => Option[List[Int]]
assert( sequence(ex(OptionMonad))(OptionMonad) == Some(List(1,2,3)) )
```

`mapM` verknüpft `sequence` und `map`. Es wird also aus `List[A]` erst `List[M[B]]` (durch `map`) und anschließend aus `List[M[B]]` (durch `sequence`) `M[List[B]]`.

```
def mapM[M[_],A,B](f : A => M[B], l: List[A])(implicit m: Monad[M]) : M[List[B]] =
  sequence(l.map(f))

assert( mapM[Option,Int,String](n => Some(n.toString), List(1,2,3))(OptionMonad)
    == Some(List("1","2","3")) )
```

`join` kann verwendet werden, um einen zweifach in einer Monade "verpackten" Wert zu "entpacken". Der Eingabetyp ist also `M[M[A]]`, der Ausgabetyp `M[A]`.

```
def join[M[_],A](x : M[M[A]])(implicit m: Monad[M]) : M[A] =
  m.bind(x, (y : M[A]) => y)

assert( join[Option,Int](Some(Some(1)))(OptionMonad) == Some(1) )
```

Weitere Monaden

Option-Monade: Die *Option-Monade* (auch *Maybe-Monade* genannt) haben wir bereits in der Einführung kennengelernt.

```
object OptionMonad extends Monad[Option] {
  override def bind[A,B](a: Option[A], f: A => Option[B]) : Option[B] = a match {
    case Some(x) => f(x)
    case None => None
  }
  override def unit[A](a: A) = Some(a)
}
```

Identitäts-Monade: Die *Identitäts-Monade* ist die einfachste Monade und entspricht normaler Funktionsapplikation. Die Übergabe der Identitäts-Monade an monadischen Code liefert den Code im gewöhnlichen Programmierstil.

```
type Id[X] = X
object IdentityMonad extends Monad[Id] {
  def bind[A,B](x: A, f: A => B) : B = f(x)
  def unit[A](a: A) : A = a
}
```

Für diese Monade wird leider nicht die Syntax der For-Comprehensions unterstützt.

Reader-Monade: Die *Reader-Monade* kodiert den "Environment Passing Style", den wir bspw. im FAE-Interpreter gesehen haben.

```
trait ReaderMonad[R] extends Monad[({type M[A] = R => A})#M] {
  override def bind[A,B](x: R => A, f: A => R => B) : R => B = r => f(x(r))(r)
  override def unit[A](a: A) : R => A = _ => a
}
```

Beim Typparameter `({type M[A] = R => A})#M` handelt es sich um eine Funktion auf Typ-Ebene, d.h. `M[A]` wird über die Gleichung `M[A] = R => A` definiert, wobei `R` ein zusätzlicher Typparameter der Monade ist. `R` wird bei jedem Funktionsaufruf als zusätzliches Argument weitergereicht, was in `M[A] = R => A` durch Currying ausgedrückt wird.

State-Monade: Bei der *State-Monade* sind Berechnungen abhängig von einem Zustand `S`, der von Berechnung zu Berechnung gereicht wird. `S` wird also beim Aufruf übergeben und mit dem Ergebnis in einem Tupel ausgegeben, was mit Currying in der Gleichung `M[A] = S => (A,S)` ausgedrückt wird.

```
trait StateMonad[S] extends Monad[({type M[A] = S => (A,S)})#M] {
  override def bind[A,B](x: S => (A,S), f: A => S => (B,S)) : S => (B,S) =
      s => x(s) match { case (a,s2) => f(a)(s2) }
  override def unit[A](a: A) : S => (A,S) = s => (a,s)
}
```

Listen-Monade: Bei der *Listen-Monade* erzeugen Berechnungen Listen von Ergebnissen, `bind` fügt jeweils alle Ergebnisse in einer Liste zusammen. In `bind` wird die Funktion `f` auf jedes Element der Liste angewendet und die Ergebnisse vom Typ `List[B]` werden zu einer Liste konkateniert.

```
object ListMonad extends Monad[List] {
  override def bind[A,B](x: List[A], f: A => List[B]) : List[B] = x.flatMap(f)
  override def unit[A](a: A) = List(a)
}
```

Continuation-Monade: Die *Continuation-Monade* kodiert CPS, es gilt `M[A] = (A => R) => R`, wobei `R` ein zusätzlicher Typparameter ist, der den Rückgabetyp von Continuations angibt.

```
trait ContinuationMonad[R] extends Monad[({type M[A] = (A => R) => R})#M] {
  type Cont[X] = (X => R) => R
  override def bind[A,B](x: Cont[A], f: A => Cont[B]) : Cont[B] =
     k => x( a => f(a)(k) )
  override def unit[A](a: A) : Cont[A] = k => k(a)
  def callcc[A,B](f: (A => Cont[B]) => Cont[A]) : Cont[A] =
    k => f( (a:A) => (_:B=>R) => k(a))(k)
}
```

Monadentransformer

In der praktischen Programmierung will man oft die Eigenschaften verschiedener Monaden kombinieren, etwa um gleichzeitig die Option-Monade und die Listen-Monade zu nutzen. Mit *Monadentransformern* ist die Komposition von Monaden möglich. Dabei handelt es sich um eine zusätzliche Fassung von jeder Monade, die mit einer weiteren Monade parametrisiert ist.

Wir verwenden wieder die Option-Monade als Beispiel und erweitern diese um eine äußere Monade:

```
type OptionT[M[_]] = { type x[A] = M[Option[A]] }

class OptionTMonad[M[_]](val m: Monad[M]) extends Monad[OptionT[M]#x] {
  override def bind[A,B](x: M[Option[A]], f: A => M[Option[B]]) : M[Option[B]] =
    m.bind(x, (z: Option[A]) => z match {
      case Some(y) => f(y)
      case None => m.unit(None)
    })
  override def unit[A](a: A) : M[Option[A]] = m.unit(Some(a))

  def lift[A](x: M[A]) : M[Option[A]] = m.bind(x, (a: A) => m.unit(Some(a)))
}
```

`lift` nimmt einen Wert vom Typ `M[A]` und macht daraus einen Wert vom Typ `M[Option[A]]`, umhüllt also den inneren Datentyp von `M` mit `Option`.

Monadischer Interpreter

Monadenbibliothek

Bevor wir verschiedene Bausteine für Interpreter anlegen, wollen wir erst eine Bibliothek von Monaden und Monadenkompositionen anlegen. Dabei orientieren wir uns an dem Stil von Standardbibliotheken für Scala oder Haskell.

```
trait Monad {
  type M[_]
  def unit[A](a: A) : M[A]
  def bind[A,B](m: M[A], f: A => M[B]) : M[B]
  implicit def monadicSyntax[A](m:M[A]) = new {
      def map[B](f: A => B) = bind(m, (x:A) => unit(f(x)))
      def flatMap[B](f: A => M[B]) : M[B] = bind(m,f)
  }
}
```

Wir definieren den Typkonstruktor `M` innerhalb der Klasse und nicht als Typparameter, da die "Typgleichungen" somit leichter ausgedrückt werden können.

Zuerst legen wir Interfaces für die Monaden an, in denen wir jeweils die Typgleichung für `M` und ggf. weitere benötigte Typen definieren. In den Interfaces legen wir auch zusätzlich benötigte Funktion neben `bind` und `unit` in abstrakter Form an. Wir definieren dann alle Monaden bis auf die Identitäts-Monade in Form von Monadentransformern, diesmal mit einer zusätzlichen inneren Monade. Dabei implementieren wir alle abstrakten Funktionen nun (unter Berücksichtigung der inneren Monade) konkret. Durch Komposition mit der Identitäts-Monade erzeugen wir effektiv eine Fassung ohne innere Monade, wodurch wir diese nicht getrennt definieren müssen.

Defunktionalisierung

Defunktionalisierung bezeichnet die Umwandlung von Higher-Order-Funktionen in First-Order-Funktionen. Diese Umwandlung ist sowohl eine Compilertechnik als auch eine Programmiertechnik.

Eine **abstrakte Maschine** bezeichnet in der theoretischen Informatik einen endlichen Automaten dessen Zustandsmenge unendlich groß sein kann.

In einem zu defunktionalisierenden Programm dürfen keine anonymen Funktionen auftreten. Um das Programm ohne anonyme Funktionen umzuschreiben, wird *Lambda Lifting* (auch *Closure Conversion* genannt) verwendet.

Lambda Lifting

Ziel von *Lambda Lifting* ist es, lokale Funktionen in Top-Level-Funktionen umzuwandeln. Lambda-Lifting kommt auch häufig in Compilern zum Einsatz, bspw. findet man im Bytecode, der beim Kompilieren von Scala-Programmen erzeugt wird, globale Funktionsdefinition für alle anonymen Funktionen im Programm.

Im folgenden Programm gibt es zwei anonyme Funktionen, nämlich `y => y*n` und `y => y+n`.

```
def map(f: Int => Int, l: List[Int]) : List[Int] = l match {
  case List() => List()
  case x::xs => f(x)::map(f,xs)
}

def addAndMulNToList(n: Int, l: List[Int]) : List[Int] =
  map(y => y*n, map(y => y+n, l))
```

Diese müssen extrahiert und als Top-Level-Funktionen definiert werden:

```
val f = (n: Int) => (y: Int) => y+n
val g = (n: Int) => (y: Int) => y*n
```

Dadurch lässt sich die Funktion `addAndMultNToList` folgendermaßen ohne anonyme Funktionen umschreiben:

```
def addAndMulNToListLL(n: Int, l: List[Int]) : List[Int] =
  map(g(n), map(f(n), l))
```

Vorgehensweise: 1. Anonyme Funktionen im Programm suchen und benennen 2. Den anonymen Funktionen entsprechende Top-Level-Funktionen anlegen 3. Code aus der ursprünglichen anonymen Funktion in die neue Funktion kopieren, freie Variablen suchen und als Parameter hinzufügen. 4. Eingabe der ursprünglichen anonymen Funktion durch Currying als weiteren Parameter ergänzen. 5. Anonyme Funktion im Programm durch Aufruf der neuen Top-Level-Funktion ersetzen 6. Lambda-Lifting im Rumpf der neuen Top-Level-Funktion fortsetzen, falls dort anonyme Funktionen auftreten

Nun wenden wir dieses Verfahren auf unseren CPS-transformierten Interpreter an. Hier ist die ursprüngliche Fassung des Interpreters, wobei alle anonymen Funktionen durch Kommentare benannt sind:

```
object CPSTransformed {
  def eval[T](e: Exp, env: Env, k: Value => T) : T = e match {
    case Num(n: Int) => k(NumV(n))
    case Id(x: String) => k(env(x))
    case Add(l: Exp, r: Exp) =>
      eval(l, env, lVal => /* addC1 */
        eval(r, env, rVal => /* addC2 */ (lVal,rVal) match {
        case (NumV(a),NumV(b)) => k(NumV(a+b))
        case _ => sys.error("Can only add numbers")
      }))
    case f@Fun(_,_) => k(ClosureV(f,env))
    case App(f,a) =>
      eval(f, env, fVal => /* appC1 */ fVal match {
        case ClosureV(Fun(p,b),cEnv) => eval(a, cEnv, aVal => /* appC2 */
          eval(b, env+(p -> aVal), k))
        case _ => sys.error("Can only apply functions")
      })
  }
}
```

Wir extrahieren die vier anonymen Funktionen und ersetzen ihre Verwendung durch Aufrufe der neuen Top-Level-Funktionen (entsprechend der obigen Schritte). Dadurch erhalten wir eine semantisch gleichbedeutende Fassung des Interpreters, in der aber keine anonymen Funktionen auftreten:

```
object LambdaLifted {
  def addC1[T](r: Exp, env: Env, k: Value => T)(lVal: Value): T =
    eval(r, env, addC2(lVal,k))
  def addC2[T](lVal: Value, k: Value => T)(rVal: Value): T = (lVal,rVal) match {
      case (NumV(a),NumV(b)) => k(NumV(a+b))
      case _ => sys.error("Can only add numbers")
    }
  def appC1[T](a: Exp, env: Env, k: Value => T)(fVal: Value) : T  = fVal match {
      case ClosureV(Fun(p,b),cEnv) => eval(a, cEnv, appC2(b,p,env,k))
      case _ => sys.error("Can only apply functions")
    }
  def appC2[T](b: Exp, p: String, env: Env, k: Value => T)(aVal: Value) : T =
    eval(b, env+(p -> aVal), k)

  def eval[T](e: Exp, env: Env, k: Value => T) : T = e match {
```

```
    case Num(n: Int) => k(NumV(n))
    case Id(x: String) => k(env(x))
    case Add(l: Exp, r: Exp) =>
      eval(l, env, addC1(r,env,k))
    case f@Fun(_,_) => k(ClosureV(f,env))
    case App(f,a) =>
      eval(f, env, appC1(a,env,k))
  }
}
```

Defunktionalisierungsschritt

In unserem ersten Beispiel liegen nach dem Lambda-Lifting noch Higher-Order-Funktionen vor, nämlich `f` und `g`.

```
val f = (n: Int) => (y: Int) => y + n
val g = (n: Int) => (y: Int) => y * n

def addAndMulNToListLL(n: Int, l: List[Int]) : List[Int] =
  map(g(n), map(f(n), l))
```

Das Programm soll nun so umgeformt werden, dass nur First-Order-Funktionen auftreten. Um den Closure nach dem ersten Currying-Schritt zu repräsentieren, legen wir einen Datencontainer an, der für beide Funktionen den Wert von `n` halten kann.

```
sealed abstract class FunctionValue
case class F(n: Int) extends FunctionValue
case class G(n: Int) extends FunctionValue
```

Außerdem legen wir eine Funktion `apply` an, die mit Instanzen von `FunctionValue` zusammen mit dem zweiten Argument aufgerufen werden kann, um den zweiten Schritt des Currying durchzuführen.

```
def apply(f: FunctionValue, y: Int) : Int = f match {
  case F(n) => y + n
  case G(n) => y * n
}

def map(f: FunctionValue, l: List[Int]) : List[Int] = l match {
  case List() => List()
  case x::xs => apply(f,x)::map(f,xs)
}

def addAndMulNToList(n: Int, l: List[Int]) : List[Int] =
  map(G(n), map(F(n), l))
```

Vorgehensweise: 1. Lege für jede Art von Higher-Order-Funktion (d.h. für jede Signatur aus Eingabe und Ausgabe) abstrakte Oberklasse an mit Unterklassen für jede verwendete Funktion des entsprechenden Typs. 2. Lege `apply`-Funktion für jede Funktionsart mit Fällen für alle Unterklassen an, kopiere Rümpfe der Top-Level-Funktionen in die Fälle. 3. Ersetze Typ von Higher-Order-Parametern mit entsprechendem Function-Value-Typ. 4. Forme Aufrufe der Higher-Order-Funktionen mit `apply` um (`f(x)` wird zu `apply(f,x)`)

Typsysteme

Ziel von Typsystemen ist es, bestimmte Arten semantischer Fehler (bspw. Addition von Funktionen oder Applikation einer Zahl) in einem syntaktisch korrekten Programm bereits vor dessen Ausführung zu erkennen. Ein Typsystem kann für die Fehler, die es erkennen soll, garantieren, dass diese bei keiner Ausführung eines

Programms auftreten. Im Gegensatz dazu kann ein Test nur die fehlerfreie Ausführung eines Programms für eine bestimmte Eingabe garantieren.

Im Kontext von Typsystemen und Typecheckern spricht man von den Eigenschaften *Soundness* und *Completeness*.

Ein Typsystem ist *sound*, wenn es jeden bei der Ausführung auftretenden Typfehler vor der Ausführung meldet, und *complete*, wenn es nur Fehler meldet, die tatsächlich bei der Ausführung auftreten. Anders ausgedrückt: Einen Typecheck, der sound ist, bestehen **nur echt typsichere** Programme und einen Typecheck, der complete ist, bestehen **alle typsicheren** Programme.

Im Fall von Soundness bestehen evtl. **echt typsichere** Programme den Typecheck nicht, im Fall von Completeness bestehen evtl. **nicht typsichere** Programme den Typecheck.

Aus dem *Satz von Rice* folgt, dass es für eine Turing-vollständige Sprache kein perfektes Typsystem, das Soundness und Completeness erfüllt, geben kann.

Satz von Rice: Sei $\mathcal{P}$ die Menge aller Turing-berechenbaren Funktionen und $\mathcal{S} \subsetneq \mathcal{P}$ eine nicht-leere, echte Teilmenge davon, so ist die Menge der Turingmaschinen, deren berechnete Funktion in $\mathcal{S}$ liegt, nicht entscheidbar.

Damit sind alle semantischen Eigenschaften von Programmen in Turing-vollständigen Sprachen nicht entscheidbar (d.h. es gibt keinen Algorithmus, der für jedes Programm entscheiden kann, ob die Eigenschaft zutrifft).

Dies wird bereits an folgendem Beispiel deutlich:

```
f()
((x: Int) => x+1) + 3
```

Um entscheiden zu können, ob im obigen Programm ein Fehler durch die Addition einer Funktion und einer Zahl entsteht, müsste entschieden werden, ob `f` terminiert – dazu müsste das Halteproblem entscheidbar sein. Da das Halteproblem unentscheidbar ist, ist durch Widerspruch bewiesen, dass kein perfektes Typsystem existieren kann. Es ist nämlich in jeder Turing-vollständigen Sprache möglich, nicht-terminierende Programme zu formulieren.

Es sind aber durchaus Typsysteme möglich, die entweder Completeness oder Soundness erfüllen. Dabei ist Soundness meist die interessantere Eigenschaft, weil damit das Typsystem die Typsicherheit garantieren kann (was für Completeness nicht der Fall ist). Es werden jedoch manche Programme abgelehnt, die in Wahrheit ohne Typfehler ausgeführt werden könnten. Es handelt sich also um eine *konservative* Abschätzung bzw. eine *Überapproximation*, da in manchen Fällen Programme "sicherheitshalber" abgelehnt werden, wenn die Typsicherheit nicht garantiert werden kann.

Für unsere Sprachen wird die Syntax jeweils durch eine *kontextfreie Grammatik* in Form der Case Classes definiert. Typkorrektheit ist aber keine kontextfreie Eigenschaft, was etwa an folgendem Programm deutlich wird:

```
wth("x", 2, Add("x",3))
```

Um hier feststellen zu können, ob `"x"` im inneren `Add`-Ausdruck gebunden ist und ob es sich dabei um eine Zahl handelt, muss der Kontext (also der `wth`-Ausdruck) betrachtet werden. Typkorrektheit ist in diesem Fall also offensichtlich eine *kontextsensitive* Eigenschaft und kann deshalb nicht direkt in der Grammatik der Sprache definiert werden. Aus diesem Grund ist ein Typsystem als zusätzlicher "Filter" notwendig, um Typkorrektheit zu garantieren. Dies gilt für die meisten Programmiersprachen.

Ein wichtiger Gesichtspunkt von Typsystemen ist auch deren Nachvollziehbarkeit für Programmierer, es muss verständlich sein, wann und warum Fehler erkannt werden. Um diese Nachvollziehbarkeit zu gewährleisten sind Typchecker meist kompositional, d.h. der Typ eines Ausdrucks ergibt sich durch die Typen seiner Unterausdrücke.

Interpreter mit Typsystem

In unseren bisherigen Interpretern haben wir beim Auftreten von Typfehlern (bspw. Addition von zwei Funktionen) einen Laufzeitfehler geworfen, wie etwa hier im FAE-Interpreter:

```
// ...
  case Add(l,r) => (eval(l,env),eval(r,env)) match {
    case (NumV(a),NumV(b)) => NumV(a+b)
    case _ => sys.error("Can only add numbers")
  }
// ...
```

Nun wollen wir für die folgende Sprache ein Typsystem definieren, so dass bereits vor der Auswertung eines Programms überprüft werden kann, ob dieses typsicher ist.

```
sealed abstract class Exp
case class Num(n: Int) extends Exp
case class Bool(b: Boolean) extends Exp
case class Add(l: Exp, r: Exp) extends Exp
case class If(c: Exp, t: Exp, f: Exp) extends Exp

def eval(e: Exp) : Exp = e match {
  case Add(l,r) => (eval(l), eval(r)) match {
    case (Num(x),Num(y)) => Num(x+y)
    case _ => sys.error("Can only add numbers")
  }
  case If(c,t,f) => eval(c) match {
    case Bool(true) => eval(t)
    case Bool(false) => eval(f)
    case _ => sys.error("Condition must be a boolean")
  }
  case _ => e
}
```

Am Interpreter ist bereits erkennbar, welche Typfehler auftreten können: Eine Additionsoperation, bei der nicht beide Summanden Zahlen sind, oder ein `If`-Ausdruck, dessen Bedingung nicht zu einem `Bool` auswertet. Um diese Fehler zu erkennen, wollen wir Werte nach Typ klassifizieren. Wir entscheiden uns dazu, zwischen Zahlen und Booleans unterscheiden.

Wir definieren eine neue abstrakte Klasse `Type` mit den zwei konkreten Unterklassen `NumType` und `BoolType`. Diese entsprechen den zwei Wertetypen, die die Auswertung eines Ausdrucks ergeben kann.

```
sealed abstract class Type
case class BoolType() extends Type
case class NumType() extends Type
```

Um Typechecking auf Programmen bzw. Ausdrücken unserer Sprache durchzuführen, definieren wir eine Funktion `typeCheck`. Bei einem Typechecker handelt es sich um eine kompositionale Zuweisung von Typen zu Ausdrücken im Programm, der Typ eines Ausdrucks wird also durch seine Unterausdrücke bestimmt. Unsere `typeCheck`-Funktion ist dementsprechend strukturell rekursiv:

```
def typeCheck(e: Exp) : Type = e match {
  case Num(_) => NumType()
  case Bool(_) => BoolType()
  case Add(l,r) => (typeCheck(l),typeCheck(r)) match {
    case (NumType(),NumType()) => NumType()
    case _ => sys.error("Type error in Add")
  }
```

```
  case If(c,t,f) => (typeCheck(c),typeCheck(t),typeCheck(f)) match {
    case (BoolType(),tType,fType) =>
      if (tType == fType) tType else sys.error("Type error in If")
    case _ => sys.error("Type error in If")
  }
}
```

Der `Num`- und `Bool`-Fall sind trivial, hier ist offensichtlich um welchen Typ es sich handelt. Im `Add`-Fall muss `typeCheck` auf beiden Unterausdrücken `NumType()` ergeben, da in unserer Sprache nur Addition von Zahlen erfolgreich ausgewertet werden kann. Sind die Summanden nicht beide vom Typ `NumType()`, so geben wir eine Fehlermeldung aus. Im `If`-Fall ist klar, dass der Typechecker auf der Bedingung `BoolType` ergeben muss, der Typ des `If`-Ausdrucks selbst lässt sich aber nicht ohne weiteres feststellen. Je nachdem, welcher Zweig betreten wird, müsste entweder der Typ von `t` oder von `f` rekursiv bestimmt und ausgegeben werden.

Um zu entscheiden, welcher Zweig betreten wird, müsste die Bedingung ausgewertet werden. Würde man diese Auswertung im Typechecker durchführen, so verliert dieser aber gewissermaßen seinen Nutzen, denn wenn er bei Typfehlern in einer `If`-Bedingung die gleichen Laufzeitfehler liefert wie der Interpreter, so bietet der Typechecker keinen Vorteil gegenüber einer Ausführung des Programms.

In einer komplexeren Sprache (z.B. der Turing-vollständigen FAE-Sprache) wäre es nicht möglich, die Bedingung ohne den Kontext des `If`-Ausdrucks auszuwerten, zudem könnte die Auswertung der Bedingung evtl. nicht terminieren, wodurch der Typechecker kein Ergebnis liefern würde. In jedem Fall verliert der Typechecker seinen Nutzen als Prüfmittel vor der eigentlichen Auswertung, sobald er Teile des Programms auswertet.

Aus diesen Gründen bleibt nur die Möglichkeit, den Typ beider Zweige zu bestimmen und auf Gleichheit zu prüfen, falls `t` und `f` den gleichen Typ besitzen, kann dieser ausgegeben werden. Dadurch wird aber bei gewissen Programmen ein Typfehler gefunden, obwohl diese fehlerfrei auswerten:

```
val ex1 = If(false,true,1)
val ex2 = Add(2, If(true,3,true))
```

Bei der Prüfung auf Gleichheit handelt es sich um eine konservative Abschätzung des Programmverhaltens – Ausdrücke, in denen im `then`- und `else`-Zweig verschiedene Typen vorliegen, werden abgelehnt, da die Typsicherheit nicht gewährleistet werden kann.

Wir können unsere Definition von Soundness für Typechecking noch etwas verfeinern: Soundness bedeutet auch, dass der Typecheck den Typ des Auswertungsergebnisses korrekt vorhersagt.

Soundness des Typsystems:

Für alle `e: Exp`, `v: Exp` und `t: Type` gilt: Falls `typeCheck(e) == t`, so gilt `eval(e) == v` mit `typeCheck(v) == t` ++oder++ `eval(e)` führt zu einem Laufzeitfehler, der nicht vom Typsystem abgedeckt wird, ++oder++ `eval(e)` terminiert nicht.

Simply-Typed Lambda Calculus (STLC)

Wir beginnen mit dem substitutionsbasierten Interpreter für den ungetypten Lambda-Kalkül (FAE), da ohne getrennte Werte (`Value`) und Closures die Implementation eines Typsystem deutlich einfacher möglich ist. Wir ergänzen Funktionen um eine Annotation des Parametertyps, die vom Interpreter ignoriert wird. Diese Sprache ist die einfachste Form des *Simply-Typed Lambda Calculus* (*STLC*).

Zusätzlich fügen wir einige gängige Erweiterungen für den STLC hinzu, nämlich ein Sprachkonstrukt `JUnit`, Bindungen mit `Let` (ohne Typannotationen), Typ-Annotationen für beliebige Ausdrücke, Produkttypen (Tupel) und Summentypen (mit zwei Alternativen):

```
sealed abstract class Type

sealed abstract class Exp
case class Num(n: Int) extends Exp
```

```
case class Id(name: String) extends Exp
case class Add(lhs: Exp, rhs: Exp) extends Exp
case class Fun(param: String, t: Type, body: Exp) extends Exp
case class App (fun: Exp, arg: Exp) extends Exp
case class JUnit() extends Exp
case class Let(x: String, xDef: Exp, body: Exp) extends Exp
case class TypeAscription(e: Exp, t: Type) extends Exp

case class Product(left: Exp, right: Exp) extends Exp
case class Fst(e: Exp) extends Exp
case class Snd(e: Exp) extends Exp

case class SumLeft(left: Exp, right: Type) extends Exp
case class SumRight(left: Type, right: Exp) extends Exp
case class EliminateSum(e: Exp, funLeft: Exp, funRight: Exp) extends Exp
```

Wir erweitern den substitutionsbasierten FAE-Interpreter um die zusätzlichen Fälle, wobei wir auch die Hilfsfunktionen `freshName`, `freeVars` und `subst` erweitern müssen.

```
def freeVars(e: Exp) : Set[String] =  e match {
  case Id(x) => Set(x)
  case Fun(p,_,b) => freeVars(b)-x
  // ...
  case JUnit() => Set.empty
  case Let(x,xDef,b) => freeVars(xDef)++(freeVars(b) - x)
  case Product(l,r) => freeVars(l)++freeVars(r)
  case SumLeft(e,_) => freeVars(e)
  case SumRight(_,e) => freeVars(e)
  case EliminateSum(e,fl,fr) => freeVars(e)++freeVars(fl)++freeVars(fr)
}

def subst(e : Exp, x: String, xDef: Exp) : Exp = e match {
  case Id(y) => if (x == y) xDef else Id(y)
  // ...
  case Fun(param,t,body) =>
    if (param == x) e else {
      val fvs = freeVars(body) ++ freeVars(xDef)
      val newvar = freshName(fvs, param)
      Fun(newvar, t,subst(subst(body, param, Id(newvar)), x, xDef))
    }
  case Let(y,ydef,body) =>
    if (x == y) Let(y,subst(ydef,x,xDef),body) else {
      val fvs = freeVars(body) ++ freeVars(xDef)
      val newvar = freshName(fvs,y)
      Let(newvar,subst(ydef,x,xDef),subst(subst(body,y,Id(newvar)),x,xDef))
    }
  case JUnit() => e
  // ...
}

def eval(e: Exp) : Exp = e match {
  case Id(x) => sys.error("Unbound identifier: " + x)
  case Add(l,r) => (eval(l), eval(r)) match {
    case (Num(x),Num(y)) => Num(x+y)
    case _ => sys.error("Can only add numbers")
```

```
  }
  case App(f,a) => eval(f) match {
    case Fun(x,_,body) => eval( subst(body,x, eval(a)))
    case _ => sys.error("Can only apply functions")
  }
  case TypeAscription(e,_) => eval(e)
  case Let(x,xdef,body) => eval(subst(body,x,eval(xdef)))
  case Product(a,b) => Product(eval(a),eval(b))
  case Fst(e) => eval(e) match {
    case Product(a,b) => a
    case _ => sys.error("Can only select first from products")
  }
  case Snd(e) => eval(e) match {
    case Product(a,b) => b
    case _ => sys.error("Can only select second from products")
  }
  case SumLeft(e,t) => SumLeft(eval(e),t)
  case SumRight(t,e) => SumRight(t,eval(e))
  case EliminateSum(e,fl,fr) => eval(e) match {
    case SumLeft(e2,_) => eval(App(fl,e2))
    case SumRight(_,e2) => eval(App(fr,e2))
    case _ => sys.error("Can only eliminate sums")
  }
  case _ => e // Num, Fun & JUnit
}
```

Wir legen ein Typsystem an, das Zahlen, Funktionen, JUnit, Produkttypen und Summentypen unterscheidet:

```
case class NumType() extends Type
case class FunType(from: Type, to: Type) extends Type
case class JUnitType() extends Type
case class ProductType(left: Type, right: Type) extends Type
case class SumType(left: Type, right: Type) extends Type
```

`FunType()`, `ProductType()` und `SumType()` sind dabei rekursiv definiert und enthalten selbst jeweils zwei `Type`-Felder. Im Fall von Funktionen sind das der Parameter- und Ausgabetyp. Die Ergänzung der Annotation für den Argumenttyp bei Funktionen sorgt dafür, dass wir den `from`-Typ von Funktionen nicht "erraten" müssen. Der `to`-Typ kann unter Angabe des `from`-Typs problemlos durch Typechecking des Funktionsrumpfes bestimmt werden.

Um mit Identifiern umzugehen, benötigt unser Typechecker ein zweites Argument, nämlich eine Typumgebung (die meist mit Γ oder *Symbol Table* bezeichnet wird), in der der Typ von Identifiern hinterlegt wird, damit er im `Id`-Fall ausgelesen werden kann.

```
def typeCheck(e: Exp, gamma: Map[String,Type]) : Type = e match {
  case Num(_) => NumType()
  case Id(x) => gamma.get(x) match {
    case Some(t) => t
    case _ => sys.error("Type error: Unbound identifier "+x)
  }
  case Add(l,r) => (typeCheck(l,gamma),typeCheck(r,gamma)) match {
    case (NumType(),NumType()) => NumType()
    case _ => sys.error("Type error: Can only add numbers")
  }
  case Fun(p,t,b) => FunType(t, typeCheck(b,gamma+(p -> t)))
  case App(f,a) => typeCheck(f,gamma) match {
```

```
    case FunType(from,to) =>
      if (from == typeCheck(a,gamma)) to
      else sys.error("Type error: Arg does not match expected type")
    case _ => sys.error("Type error: Left expression must be a function")
  }
  case JUnit() => JUnitType()
  case Let(x,xDef,b) => typeCheck(b,gamma+(x -> typeCheck(xDef,gamma)))
  case TypeAscription(e,t) =>
    if (typeCheck(e,gamma)==t) t
    else sys.error("Type error: Type does not match")
  case Product(l,r) => ProductType(typeCheck(l,gamma),typeCheck(r,gamma))
  case Fst(e) => typeCheck(e,gamma) match {
    case ProductType(l,_) => l
    case _ => sys.error("Type error: Can only project products")
  }
  case Snd(e) => typeCheck(e,gamma) match {
    case ProductType(_,r) => r
    case _ => sys.error("Type error: Can only project products")
  }
  case SumLeft(l,t) => SumType(typeCheck(l,gamma),t)
  case SumRight(t,r) => SumType(t,typeCheck(r,gamma))
  case EliminateSum(c,fl,fr) => typeCheck(c,gamma) match {
    case SumType(l,r) => (typeCheck(fl,gamma),typeCheck(fr,gamma)) match {
      case (FunType(lf,lt),FunType(rf,rt)) if l==lf && r==rf && lt==rt =>
        if (lTo==rTo) lTo
        else sys.error("Type error: Functions must have same return type")
      case _ => sys.error("Type error: 2nd and 3rd arg must be functions")
    }
    case _ => sys.error("Type error: Can only eliminate sums")
  }
}
```

Soundness und Terminierung von STLC: Für `e: Exp` mit `typeCheck(e) == t` gilt: `eval(e)` terminiert und `typeCheck(eval(e),Map()) == t`.

STLC ist nicht Turing-vollständig und in STLC können nur terminierende Programme verfasst werden. Aus diesem Grund wird STLC bei der Implementation von Programmiersprachen häufig auf Typ-Level verwendet, da man bspw. Typfunktionen und deren Applikation formulieren will, aber nicht-terminierende Ausdrücke auf Typ-Level verhindern muss, da sonst der Typechecker nicht mehr zwingend terminiert.

Hindley-Milner-Typinferenz

Zuletzt wollen wir noch ein Typsystem betrachten, das ohne jegliche Typannotationen auskommt und bei dem Typen *inferiert* werden können. Dazu wird beim Typechecking eine Liste von *Constraints* erzeugt und ähnlich wie bei einem linearen Gleichungssystem nach einer Belegung mit Typen gesucht, die alle Constraints erfüllt.

Wir kehren dazu zum nicht-erweiterten STLC zurück, wobei wir wieder den substitutionsbasierten Interpreter verwenden. Wir unterscheiden die Typen `FunType()` und `NumType()`, außerdem ergänzen wir Typvariablen.

```
sealed abstract class Type
case class FunType(from: Type, to: Type) extends Type
case class NumType() extends Type
case class TypeVar(x: String) extends Type
```

Da der Typ von Identifiern (d.h. Funktionsparametern) erst bei deren Verwendung bestimmt werden kann,

wird Identifiern bei ihrem ersten Auftreten eine Typvariable zugeordnet. Bei der Auswertung von Ausdrücken, in dem ein Identifier genutzt wird, können dann Constraints für die Typvariable erzeugt werden – wird ein Identifier bspw. mit einer Zahl addiert, so muss er den Typ `NumType()` besitzen.

Bei den Constraints handelt es sich um Typgleichungen, bei denen der linke und rechte Typ übereinstimmen müssen. Diese werden als Tupel der Form `(Type,Type)` repräsentiert. Die Ausgabe von `typeCheck` ist eine Liste solcher Constraint-Tupel zusammen mit dem Typen des Ausdrucks `e`, bei dem es sich auch um eine Typvariable handeln kann.

Um Identifier mit ihrer jeweiligen Typvariable zu assoziieren, verwenden wir wieder eine Typumgebung Γ (`g`).

```
var typeVarCount: Int = 0
def freshTypeVar() : Type = {
  typeVarCount += 1
  TypeVar("X"+typeVarCount.toString)
}

def typeCheck(e: Exp, g: Map[String,Type]) : (List[(Type,Type)],Type) = e match {
  case Num(_) => (List(),NumType())
  case Id(x) => g.get(x) match {
    case Some(t) => (List(),t)
    case _ => sys.error("Unbound identifier: "+x)
  }
  case Add(l,r) => (typeCheck(l,g),typeCheck(r,g)) match {
    case ((lEqs,lt),(rEqs,rt)) =>
      (lt->NumType() :: rt->NumType() :: lEqs ++ rEqs, NumType())
  }
  case Fun(p,b) =>
    val xt = freshTypeVar()
    val resBody = typeCheck(b,g+(p->xt))
    (resBody._1, FunType(xt,resBody._2))
  case App(f,a) =>
    val toType = freshTypeVar()
    (typeCheck(f,g),typeCheck(a,g)) match {
      case ((fEqs, ft), (aEqs, at)) =>
        ((ft, FunType(at, toType)) :: fEqs ++ aEqs, toType)
    }
}
```

Im `Num`-Fall wird eine leere Constraint-Liste und der Typ `NumType()` zurückgegeben. Im `Id`-Fall wird der Identifier `x` in der Typumgebung nachgeschlagen, wird ein Eintrag gefunden, so wird der entsprechende Typ und eine leere Constraint-Liste ausgegeben, ansonsten wird ein Fehler geworfen.

Im `Add`-Fall wird erst Typechecking auf den Unterausdrücken durchgeführt, die Constraint-Liste wird um zwei Gleichungen erweitert: Der Typ beider Unterausdrücke muss mit `NumType()` übereinstimmen, da nur Zahlen addiert werden können. Es wird der Typ `NumType()` ausgegeben.

Im `Fun`-Fall wird eine "frische" Typvariable für den `param`-Identifier erzeugt, `typeCheck` wird rekursiv auf dem Rumpf aufgerufen, wobei in der Typumgebung der Parameter an die neue Typvariable gebunden wird. Das Typechecking des Rumpfes ergibt die Constraint-Liste und den `to`-Typ der Funktion, für den `from`-Typ wird die neue Typvariable eingesetzt.

Im `App`-Fall wird eine neue Typvariable `toType` für das Ergebnis der Funktionsapplikation erzeugt, anschließend wird rekursiv Typechecking auf beiden Unterausdrücken durchgeführt. Die Constraint-Liste wird erweitert um die Bedingung, dass der Typ der Funktion `FunType()` ist, wobei der `from`-Typ mit dem Typ des Arguments und der `to`-Typ mit `toType` übereinstimmt. Es wird `toType` ausgegeben.

Unifikationsalgorithmus von Robinson

Die `typeCheck`-Funktion trifft im Gegensatz zu der im STLC-Typsystem (bis auf das Erkennen ungebundener Identifier) noch gar keine Aussage darüber, ob ein Ausdruck typkorrekt ist. Dazu muss für die von `typeCheck` ausgegebene Liste von Typgleichungen geprüft werden, ob sich alle Gleichungen *unifizieren* lassen. Dazu verwenden wir den *Unifikationsalgorithmus nach Robinson*:

```
def substitution(x: String, s: Type): Type => Type = new Function[Type,Type] {
  def apply(t: Type) : Type = t match {
    case FunType(from, to) => FunType(this(from), this(to))
    case NumType() => NumType()
    case TypeVar(y) => if (x==y) s else t
  }
}

def freeTypeVars(t: Type) : Set[String] = t match {
  case FunType(f,t) => freeTypeVars(f)++freeTypeVars(t)
  case NumType() => Set()
  case TypeVar(x) => Set(x)
}

def unify(eq: List[(Type,Type)]) : Type => Type = eq match {
  case List() => identity
  case (NumType(),NumType())::rest => unify(rest)
  case (FunType(f1,t1),FunType(f2,t2))::rest => unify(f1->f2::t1->t2::rest)
  case (TypeVar(x1),TypeVar(x2))::rest if x1==x2 => unify(rest)
  case (TypeVar(x),t)::rest =>
    if (freeTypeVars(t)(x)) sys.error(s"Occurs check: $x occurs in $t")
    val s = substitution(x,t)
    s.andThen(unify(rest.map(tup => (s(tup._1),s(tup._2)))))
  case (t,TypeVar(x))::rest => unify((TypeVar(x),t)::rest)
  case (t1,t2)::_ => sys.error(s"Cannot unify $t1 and $t2")
}
```

`unify` bestimmt die einfachste Belegung der Typvariablen, die alle durch `typeCheck` generierten Gleichungen erfüllt, und gibt eine Funktion aus, die alle Typvariablen entsprechend dieser Belegung substituiert.

Stimmen im ersten Tupel der Liste von Gleichungen die linke und rechte Seite bereits überein, so wird `unify` einfach auf der Restliste aufgerufen. Steht links und rechts der Typ `FunType()`, so wird die Liste um zwei Bedingungen erweitert, nämlich dass der `from`- und `to`-Typ jeweils übereinstimmen. Hierbei handelt es sich um *generative Rekursion*.

Steht im ersten Tupel links eine Typvariable und rechts ein anderer Typ, so wird mittels `freeTypeVars` geprüft, ob die Typvariable innerhalb des anderen Typs auftritt. Ist dies der Fall, so ist die Gleichung "selbstbezüglich" und kann unmöglich erfüllt werden (bspw. gibt es keinen Typ `X`, für den `X` und `FunType(X,X)` gleichbedeutend sind). In diesem Fall wird also ein Fehler geworfen.

Andernfalls wird mit `substitution` eine Funktion angelegt, die alle Vorkommen der (linken) Typvariable durch den anderen (rechten) Typ ersetzt. Mit `andThen` wird eine Komposition der Substitution nach der Ergebnisfunktion des rekursiven Aufrufs von `unify` angelegt, wobei `unify` auf der Restliste aufgerufen wird, in der die Substitution mit `map` auf alle Tupel angewendet wurde.

Steht die Typvariable rechts und ein anderer Typ links, so werden die Einträge im Tupel vertauscht, in allen anderen Fällen können die zwei Typen im Tupel nicht unifiziert werden und es wird ein Fehler geworfen.

Erreicht `unify` das Ende der Liste, so wird die Identitätsfunktion ausgegeben, durch die Komposition der Substitutionsfunktionen gibt `unify` also eine Funktion aus, die die einen Typ akzeptiert darin der

unifizierenden Belegung entsprechend alle Typvariablen substituiert. Es kann also der gesamte Typecheck mit der folgenden Funktion durchgeführt werden:

```
def doTypeCheck(e: Exp): Type = {
  val (eqs,t) = typeCheck(e,Map())
  unify(eqs)(t)
}
```

Completeness der Typinferenz: Gibt es für ein Programm Typannotationen, mit denen es vom STLC-Typechecker akzeptiert wird, so akzeptiert der Typecheck mit Typinferenz auch die nicht-typannotierte Variante des Programms.

Soundness: Für alle `e: Exp` gilt: Falls `doTypeCheck(e) == t`, dann gilt `eval(e) == v` mit `doTypeCheck(v) == t` (unter α-Konversion, d.h. Umbennenung der Typvariablen ohne Veränderung der Bedeutung).

Let-Polymorphismus

Mit der bisherigen Definition der Sprache und des Typsystems ist es nicht möglich, *polymorphe* Funktionen auszudrücken, die für mehrere Argumenttypen genutzt werden können. Bspw. müsste man die Identitätsfunktion oder die Kompositionsfunktion für jeden Eingabetyp getrennt anlegen.

Wir ermöglichen polymorphe Funktionen durch lokale Bindungen mit `Let`:

```
case class Let(x: String, xDef: Exp, body: Exp) extends Exp
```

In `typeCheck` ergänzen wir den folgenden Fall für `Let`:

```
def typeCheck(e: Exp, g: Map[String,Type]) : (List[(Type,Type)],Type) = e match {
  // ...
  case Let(x,xDef,b) =>
    val (eqs1,_) = typeCheck(xDef,g)
    val (eqs2,t) = typeCheck(subst(b,x,xDef),g)
    (eqs1++eqs2,t)
}
```

Erst wird `typeCheck` rekursiv auf `xDef` aufgerufen, anschließend rufen wir `typeCheck` rekursiv auf dem Rumpf des `Let`-Ausdrucks auf. Dabei verwenden wir aber (innerhalb von `typeCheck`!) die `subst`-Funktion des Interpreters, um alle Vorkommen von `x` im Rumpf durch `xDef` zu ersetzen. Es werden die Constraints von `typeCheck` auf `xDef` und auf dem Rumpf konkateniert und mit dem Ergebnistyp von `typeCheck` auf dem Rumpf ausgegeben.

Durch die Substitution vor der Durchführung des Typechecks wird jedes Vorkommen von `x` unabhängig von den anderen Vorkommen auf Typkorrektheit geprüft und die Typen der Vorkommen müssen nicht zwingend übereinstimmen.

Bspw. würde also beim Typechecking des Ausdrucks `Let("id",Fun("x","x"),...)` das Typechecking für jedes Vorkommen der Identitätsfunktion unabhängig voneinander durchgeführt werden. Dadurch kann die Funktion polymorph für jeden beliebigen Argumenttyp genutzt werden, da für alle Vorkommen jeweils der korrekte Typ (etwa `FunType(NumType(),NumType())`) inferiert wird.