Die Aufgabe der **Typinferenz** ist es, eine geeignete Typannahme zu raten bzw. konstruktiv zu finden. Dies ist

- unproblematisch für Parameter
- schwieriger für Typschemata (Typen der Funktionen), daher fordert man dann weitere Einschränkungen, die allerdings nur bei komplexen Beispielen relevant werden.

Beispiel 5.22 (Notwendigkeit der Typannahme)**.** Wir betrachten das folgende Programm:

```
f :: [a] → [a]
f xs = if length xs == 0 then fst (g xs xs) else xs

g :: [a] → [b] → ([a],[b])
g xs ys = (f xs, f ys)

h = g [3,4] [True, False]

fst (x,y) = x   -- first
```

Bezüglich der Wohlgetyptheit stellen wir Folgendes fest:

- Mit Typangaben für `f` und `g` ist das Beispiel wohlgetypt.
- Ohne Typangaben für `f` und `g` ist das Beispiel nicht wohlgetypt, da der Typ

  ```
  g :: [a] → [a] → ([a], [a])
  ```

  hergeleitet wird.

Das **Prinzip bei der Typinferenz** ist das Folgende: Innerhalb von rekursiven Aufrufen/ Definitionen haben Funktionen kein Typschema, sondern nur einen Typ, anderenfalls wäre die Typinferenz unentscheidbar.

5.6 Funktionale Konstrukte in imperativen Sprachen

In diesem Kapitel haben wir die folgenden charakteristischen Eigenschaften funktionaler Programmiersprachen diskutiert:

- referenzielle Transparenz
- algebraische Datentypen und Pattern Matching
- Funktionen höherer Ordnung
- parametrischer Polymorphismus
- lazy-Auswertung

Der erste und der letzte Punkt sind schwer mit imperativen Programmiersprachen zu kombinieren, da diese ein anderes Auswertungsprinzip erfordern. Die anderen Aspekte sind oder können teilweise in imperativen Programmiersprachen integriert werden. So wurde schon bald nach der erfolgreichen Verbreitung von Java mit der Sprache Pizza[3] (Odersky and Wadler, 1997) ein Ansatz vorgestellt, wie diese Aspekte in Java integriert werden können. Pizza selbst hat keine weite Verbreitung gefunden, allerdings haben die Ideen dieser Erweiterung inzwischen in Java bzw. anderen Sprachen Einzug gehalten:

5.6.1 Parametrischer Polymorphismus

Wie wir schon diskutiert haben, ist parametrischer Polymorphismus in Form von generischen Modulen oder parametrisierten Datentypen auch schon in anderen Sprachen existent. Die Kombination von parametrischem Polymorphismus mit Untertypen und Zuweisungen birgt allerdings einige Probleme, so dass es bis zur Version 5 von Java gedauert hat, bis auch dieses Konzept in Java eingeführt wurde (vgl. Kapitel 4.5).

Ein wichtiger Aspekt funktionaler Sprachen, nämlich die automatische Typinferenz, ist in vielen streng getypten imperativen Sprachen noch wenig verbreitet. Eine positive Ausnahme hiervon ist die Programmiersprache Scala[4], die die Ideen von Pizza aufgegriffen hat (was nicht verwunderlich ist, weil der Hauptentwickler von Scala, Martin Odersky, auch Pizza entwickelt hat).

Scala ist wie Pizza ebenfalls ein Ansatz, die Eigenschaften objektorientierter und funktionaler Sprachen zu integrieren. Diese Motivation wird auch dadurch sichtbar, dass Scala zwischen veränderbaren und unveränderbaren Objekten („variables“ bzw. „values“) unterscheidet, die mit unterschiedlichen Schlüsselworten (`var` bzw. `val`) eingeführt werden. Die folgende Deklaration führt eine unveränderbare Variable `x` ein. Hierbei wird der Typ automatisch inferiert:

[3] `http://pizzacompiler.sourceforge.net/`
[4] `http://www.scala-lang.org`

```
scala> val x = 5
x: Int = 5

scala> x * 2
res0: Int = 10

scala> x = 4
<console>:8: error: reassignment to val
```

Veränderbaren Variablen kann man dagegen wie üblich neue Werte zuweisen:

```
scala> var y = "Hello"
y: java.lang.String = Hello

scala> y = y + " World!"
y: java.lang.String = Hello World!

scala> println(y)
Hello World!
```

Bei Funktionen ist allerdings die Typinferenz nicht so umfassend wie in funktionalen Sprachen, denn hier müssen zumindest die Typen der Parameter angegeben werden. Funktionen werden durch das Schlüsselwort `def` deklariert und einzeilige Funktionsdeklarationen können ohne geschweifte Klammern um den Rumpf erfolgen:

```
scala> def inc(x:Int) = x+1
inc: (x: Int)Int

scala> inc(5)
res1: Int = 6
```

Wie man sieht, wird zumindest der Ergebnistyp inferiert.

Parametrische Klassen sind ähnlich wie in Java 5 möglich. Attribute von polymorphem Typ können als Klassenparameter festgelegt werden, sodass deren Wert direkt bei der Erzeugung von Objekten festgelegt wird:

```
class List[T](elem: T, next: List[T]) {
  def printList():Unit = {
    println(elem)
    if (next != null) next.printList()
  }
}
```

Durch diese Festlegung müssen die Typen bei der Erzeugung nicht angegeben werden:

```
scala> val xs = new List(1, new List(2, new List(3, null)))
```

```
xs: List[Int] = List@135572b
```

Scala unterstützt wie funktionale Sprachen kompakte Notationen. So kann man leere Argumentlisten beim Aufruf weglassen:

```
scala> xs.printList()
1
2
3

scala> xs.printList
1
2
3

scala> xs printList
1
2
3
```

Das letzte Beispiel zeigt, dass man auch den Punkt bei Zugriff auf Methoden weglassen kann. Dies wird ausgenutzt, um auch Operatoren einfach als Methoden aufzufassen, bei denen der Punkt fehlt:

```
scala> 1+2
res2: Int = 3

scala> (1).+(2)
res3: Int = 3
```

5.6.2 Funktionen höherer Ordnung

Viele imperative Programmiersprachen bieten auch Funktionen höherer Ordnung an, allerdings häufig auch eingeschränkter als in funktionalen Sprachen (z. B. ohne Currying). So lässt schon Pascal Funktionen als Parameter zu, allerdings in einer typunsicheren Weise, da keine Parametertypen angegeben werden. Ähnlich dazu lässt auch C Funktionszeiger zu. Scala unterstützt auch Definitionen ähnlich wie Lambda-Abstraktionen:

```
scala> val inc = (x:Int) => x + 1
inc: Int => Int = <function1>
```

Die Parametertypen können dabei weggelassen werden, wenn diese aus dem Kontext inferiert werden können:

```
scala> val num14 = List(1,2,3,4)
```

```
num14: List[Int] = List(1, 2, 3, 4)

scala> num14.filter((x:Int) => x > 1)
res0: List[Int] = List(2, 3, 4)

scala> num14.filter((x) => x > 1)
res1: List[Int] = List(2, 3, 4)

scala> num14.filter(x => x > 1)
res2: List[Int] = List(2, 3, 4)

scala> num14.filter(_ > 1)
res3: List[Int] = List(2, 3, 4)
```

Beim letzten Ausdruck wird die „Platzhalter"-Syntax ausgenutzt, bei der die Funktionsparameter in einem Ausdruck durch Unterstriche gekennzeichnet werden. Durch diese Syntax unterstützt Scala auch die partielle Applikation, d.h. die Anwendung von Funktionen ohne alle Argumente:

```
scala> def sum(a: Int, b: Int) = a + b
sum: (a: Int, b: Int)Int

scala> def inc = sum(1,_:Int)
inc: Int => Int

scala> inc(4)
res0: Int = 5
```

Scala erlaubt die Definition Funktionen höherer Ordnung in der üblichen Syntax (allerdings mit notwendigen Typangaben für die Parameter):

```
scala> def twice(f: Int => Int, x: Int) = f(f(x))
twice: (f: Int => Int, x: Int)Int

scala> twice(inc,3)
res1: Int = 5
```

Ähnlich wie bei Klassen können diese auch über den Typ parametrisiert werden:

```
scala> def twice[A](f: A => A, x: A) = f(f(x))
twice: [A](f: A => A, x: A)A

scala> twice(inc,3)
res1: Int = 5
```

```
scala> twice((x:Boolean) => !x, true)
res2: Boolean = true
```

5.6.3 Parameterübergabe

Die Parameterübergabe bei Scala ist call-by-value, wie man im folgenden Beispiel sehen kann:

```
scala> def byValue(x: Int, y: Int) = if (x==0) y else x
byValue: (x: Int, y: Int)Int

scala> byValue(1,1/0)
java.lang.ArithmeticException: / by zero
```

Wie wir zuvor gesehen haben, kann man die Namensübergabe dadurch realisieren, dass man die Parameter als Prozeduren ohne Parameter übergibt. Dies wird in Scala implizit durch eine spezielle Syntax unterstützt, indem man vor die Typen der Namensparameter „=>“ schreibt:

```
scala> def byName(x: Int, y: =>Int) = if (x==0) y else x
byName: (x: Int, y: => Int)Int

scala> byName(1,1/0)
res0: Int = 1
```

5.6.4 Algebraische Datentypen und Pattern Matching

Scala unterstützt einfaches Pattern Matching mit einem `match`-Konstrukt:

```
y match {
  case 0  => "null"
  case 42 => "meaning of life"
  case _  => "something else"
 }
```

Algebraische Datentypen wie in Haskell werden zwar nicht direkt unterstützt, aber Scala nutzt die Ähnlichkeit von Vererbung und Vereinigungstypen aus, d. h. die verschiedenen Konstruktoren eines Typs können als Unterklassen eines gemeinsamen Obertyps definiert werden. Diese kann man mittels Pattern Matching unterscheiden, wenn man diese als „`case`-Klassen“ definiert. Zusätzlich bieten „`case`-Klassen“ eine einfache Konstruktion ohne „`new`“ an, wie das folgende Beispiel zeigt:

```
// Ausdrücke sind Zahlen oder binäre Operatoren mit Ausdrücken:
abstract class Expr
case class Num(num: Int) extends Expr
```

```
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr

// Berechne den Wert eines Ausdrucks durch Pattern Matching:
def expValue(expr: Expr): Int = expr match {
  case Num(n) => n
  case BinOp("+",e1,e2) => expValue(e1) + expValue(e2)
  case BinOp("*",e1,e2) => expValue(e1) * expValue(e2)
}

val exp = BinOp("+",Num(2),BinOp("*",Num(2),Num(3)))

println(expValue(exp))
```

Beim Pattern Matching von case-Klassen wird die Vererbung durch den syntaktischen Zucker noch weitgehend versteckt. Grundsätzlich bietet Scala jedoch auch die Möglichkeit Pattern Matching über den Typ des Arguments vorzunehmen, wie folgendes Beispiel zeigt:

```
def generalSize(x: Any) = x match {
  case s: String    => s.length
  case m: Map[_, _] => m.size
  case _            => -1
}
```

Darüber hinaus bietet Scala noch viele weitere Sprachkonzepte an, auf die wir hier aber nicht eingehen. Wir können aber festhalten:

- Prinzipiell ist die Integration vieler Aspekte funktionaler Programmierung auch in imperativen Sprachen möglich.
- Viele Vorteile funktionaler Programmiersprachen (Verständlichkeit durch referenzielle Transparenz, kurze und präzise Definitionen durch musterorientierte Regeln, Funktionen höherer Ordnung, Typinferenz) sind aber nur in rein funktionalen Programmiersprachen wirklich ausnutzbar.