Fortgeschrittene Programmierung

Prof. Dr. Michael Hanus, Frank Huch[1][2]

8. Oktober 2009

[1] TEXnische Umsetzung: Nick Prühs
[2] Teilweise durchgesehen von: Sebastian Fischer

Inhaltsverzeichnis

Kapitel 1

Java Generics

1.1 Einführung

Seit der Version 5.0 (2004 veröffentlicht) bietet Java die Möglichkeit der *generischen Programmierung*: Klassen und Methoden können mit Typen parametrisiert werden. Hierdurch eröffnen sich ähnliche Möglichkeiten wie mit Templates in C++.

Als einfaches Beispiel betrachten wir eine sehr einfache Containerklasse, welche keinen oder einen Wert speichern kann. In Java könnte das z.B. wie folgt definiert werden:

```
public class Maybe {
    private Object value;
    private boolean empty;


    public Maybe() { empty = true; }

    public Maybe(Object v) {
        value = v;
        empty = false;
    }


    public boolean isEmpty() {
        return empty;
    }

    public Object fromMaybe() {
        if (!empty) {
            return value;
        }
```

```
        throw new MaybeException();
    }
}
```

Wir nehmen an, `MaybeException` sei abgeleitet von `RuntimeException`.

Die Verwendung der Klasse könnte dann aussehen wie folgt:

```
Maybe mv = new Maybe(new Integer(42));

...

if (!mv.isEmpty()) {
    Integer n = (Integer)mv.fromMaybe();
}
```

Der Zugriff auf das gespeicherte Objekt der Containerklasse erfordert also jedes Mal explizite Typcasts. Es ist *keine Typsicherheit* gegeben: Im Falle eines falschen Casts tritt eine `ClassCastException` zur Laufzeit auf.

Beachte: Die Definition der Klasse Maybe ist völlig unabhängig vom Typ des gespeicherten Wertes; der Typ von value ist `Object`. Wir können also beliebige Typen erlauben und in der Definition der Klasse von diesen abstrahieren: Das übergeben des Types als Parameter nennt man *parametrisierter Polymorphismus*.

Die Klasse lautet dann wie folgt:

```
public class MayBe<T> {
    private T value;
    private boolean empty;


    public Maybe() { empty = true; }

    public Maybe(T v) {
        value = v;
        empty = false;
    }


    public boolean isEmpty() {
        return empty;
    }
```

```
    public T fromMaybe() {
        if (!empty) {
            return value;
        }

        throw new MaybeException();
    }
}
```

Die Verwendung der Klasse sieht dann so aus:

```
Maybe<Integer> mv = new Maybe<Integer>(new Integer(42));

...

if (!mv.isEmpty()) {
    Integer n = (mv.fromMaybe();
}
```

Es sind also keine expliziten Typcasts mehr erforderlich, und ein Ausdruck wie

```
mv = new Maybe<Maybe<Integer>>(mv);
```

liefert nun bereits eine Typfehlermeldung zur Compilezeit. Natürlich sind auch mehrere Typparameter erlaubt:

```
public class Pair<A, B> {
    private A first;
    private B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public A getFirst() { return first; }
    public B getSecond() { return second; }
}
```

1.2 Zusammenspiel mit Generics

Wir wollen unsere Klasse `Maybe` so erweitern, dass sie auch das Interface Comparable implementiert:

```
public class Maybe<T extends Comparable<T>> implements Comparable<Maybe<T>> {

    ...

    public int compareTo(Maybe<T> o) {
        if (empty) {
            if (o.isEmpty()) {
                return 0;
            } else {
                return -1;
            }
        } else {
            if (o.isEmpty()) {
                return 1;
            } else {
                return value.compareTo(o.fromMaybe());
            }
        }
    }
}
```

Hierbei wird sowohl für Interfaces als auch für echte Vererbung das Schlüsselwort `extends` verwendet. Wir können auch mehrere Einschränkungen an Typvariablen aufgezählen. Für drei Typparameter sieht das aus wie folgt:

```
<T extends A<T>, S, U extends B<T, S>>
```

Hier muss `T` die Methoden von `A<T>` und `U` die Methoden von `B<T,S>` zur Verfügung stellen.

1.3 Wildcards

Die Klasse `Integer` ist Unterklasse der Klasse `Number`. Somit sollte doch auch der folgende Code möglich sein:

```
Maybe<Integer> mi = new Maybe<Integer>(42);
Maybe<Number> mn = mi;
```

Das Typsystem erlaubt dies aber nicht. Das Problem wird deutlich, wenn wir zur Klasse `Maybe` noch eine Setter-Methode hinzufügen:

```
public set(T v) {
    empty = false;
    value = v;
}
```

Dann wäre auch folgendes möglich:

```
mn.set(new Float(42));
Integer i = mi.fromMaybe();
```

Ein Widerspruch, denn `mi` und `mn` bezeichnen die gleichen Objekte. Dennoch benötigt man manchmal einen Supertyp für polymorphe Klassen, also einen Typ, der alle anderen Typen umfasst: So beschreibt

```
Maybe<?> mx = mi;
```

einen Maybe von unbekanntem Typ. `Maybe<?>` repräsentiert alle anderen Maybe-Instantiierungen, z.B. `Maybe<Integer>` oder `Maybe<Maybe<Object»`.

Damit können aber nur Methoden verwendet werden, die für jeden Typ passen, also z.B.

```
Object o = mx.fromMaybe();
```

Können wir auch die Methode `set(T)` aufrufen? Hierzu benötigen wir einen Wert, der zu allen Typen gehört: `null`.

```
mx.set(null);
```

Andere `set`-Aufrufe sind nicht möglich.

Der Wildcardtyp `?` ist aber leider oft nicht ausreichend, z.B. wenn man in einer Collection auch verschiedene Untertypen speichert, wie GUI-Elemente. Dann kann man sog. *beschränkte Wildcards* verwenden:

`<? extends A>` für *alle* Untertypen des Typs A *(Kovarianz)*
`<? super A>` für *alle* Obertypen von A *(Kontravarianz)*

Es folgen einige Beispiele.

Ausdruck	funktioniert?	Begründung
`Maybe<? extends Number> mn = mi;`		
`Integer i = mn.fromMaybe();`	nein!	`?` könnte auch `Float` sein
`Number n = mn.fromMaybe();`	ja	
`mn.set(n);`	nein!	`?` könnte spezialiserter als `Number` sein
`mn.set(new Integer(42));`	nein!	`?` könnte auch `Float` sein
`mn.set(null);`	ja	
`Maybe<? super Integer> mx = mi;`		
`Integer i = mx.fromMaybe();`	nein!	`?` könnte auch `Number` oder `Object` sein
`Number n = mx.fromMaybe();`	nein!	`?` könnte auch `Object` sein
`Object o = mx.fromMaybe();`	ja	
`mx.set(o);`	nein!	`?` könnte auch `Number` sein
`mx.set(n);`	nein!	`n` könnte auch vom Typ `Float` sein
`mx.set(i);`	ja	

Tabelle 1.1: Ausdrücke mit Wildcards

Wir können also aus einem `Maybe<? extends A>` Objekte vom Typ `A` herausholen, und in einen `Maybe<? super A>` Objekte vom Typ `A` hereinstecken.

Im folgenden betrachten wir noch ein paar Erweiterungen der Klasse `Maybe`. Gegeben sei folgendes Interface für einen Wrapper, der es erlaubt, Funktionen als Objekte zu speichern:

```
interface Fun<arg, res> {
    public res apply(arg a);
}
```

Für konkrete Funktionen erstellen wir einfach Klassen, die obiges Interface implementieren. Nun können wir unsere Klasse `Maybe` erweitern wie folgt:

```
public T fromMaybeWithDefault(T default) {
    return (empty ? default : value);
}

public void map(Fun<T, T> f) {
    if (!empty) {
        value = f.apply(value);
    }
}
```

Die Verwendung unserer erweiterten Klasse könnte wie folgt aussehen:

```
Maybe<Integer> mi = new Maybe<Integer>(42);
mi.map(new Fun<Integer, Integer>() {
    public Integer apply(Integer n) {
        return n + 1;
    }
});
```

Die Konzepte der Funktionen `fromMaybeWithDefault(T)` und `map(Fun<T, T>)` lassen sich sogar vereinen:

```
public <resT> resT applyWithDefault(resT default, Fun<T, resT> f) {
    if (empty) {
        return default;
    } else {
        return f.apply(value);
    }
}
```

Die Verwendung der neuen Methode könnte so aussehen:

```
Boolean b = mi.applyWithDefault(true,
    new Fun<Integer, Boolean) {
        public Boolean apply(Integer n) {
            return (n >= 42);
        }
    });
```

Beachte: Die Verwendung von Funktionsobjekten `Fun` wird erst durch Typparameter sinnvoll, da sonst keinerlei Typsicherheit gewährleistet werden kann, ohne viele spezielle Interfaces für einzelne Verwendungen zu definieren.

Für jede generische Klasse wird dabei genau eine ungetypte Variable erzeugt, bei der statt der Typvariablen die speziellsten möglichen Typen verwendet werden. Im Gegensatz dazu werden die Templates in C++ für jede Verwendung spezialisiert, d.h. es werden Klassen `MaybeInteger`, `MaybeNumber` oder `MaybeBoolean` erzeugt.

Vorteile der Java-Lösung:

- weniger Code
- Codeerzeugung unabhängig von Klassenverwendung

Nachteil der Java-Lösung:

- keine Typinstanziierung durch primitive Typen wie `int` oder `float`

Kapitel 2

Nebenläufige Programmierung in Java

2.1 Allgemeine Vorbemerkungen

2.1.1 Motivation

Wozu brauchen wir nebenläufige Programmierung in Java? Häufig möchten wir, dass eine Anwendung mehrere Aufgaben übernehmen soll. Gleichzeitig soll die *Reaktivität* der Anwendung erhalten bleiben. Beispiele für solche Anwendungen sind:

- GUIs
- Betriebssystemroutinen
- verteilte Applikationen (Webserver, Chat, ...)

2.1.2 Lösung

Dies erreichen wir mittels *Nebenläufigkeit (Concurrency)*. Durch die Verwendung von Threads bzw. Prozessen können einzelne Aufgaben einer Anwendung unabhängig von anderen Aufgaben programmiert und ausgeführt werden.

2.1.3 Weitere Begriffe

Parallelität. Durch die parallele Ausführung mehrerer Prozesse soll eine schnellere Ausführung erreicht werden (High-Performance-Computing).

Verteiltes System. Mehrere Komponenten in einem Netzwerk arbeiten zusammen an einem Problem. Meist gibt es dabei eine verteilte Aufgabenstellung, manchmal nutzt man verteilte Systeme auch zur Parallelisierung.

2.1.4 Arten von Multitasking

Die Prozessorzeit wird durch Scheduler auf die nebenläufigen Threads bzw. Prozesse verteilt. Wir unterscheiden zwei Arten von Multitasking:

1. *Kooperatives Multitasking*: Ein Thread rechnet so lange, bis er die Kontrolle wieder abgibt (z.B. mit `yield()`) oder auf Nachricht `suspend()`. In Java finden wir dies bei den sog. *Green threads*.

2. *Präemtives Multitasking*: Der Scheduler kann Tasks auch die Kontrolle entziehen. Hierbei genießen wir oft mehr Programmierkomfort, da wir uns nicht so viele Gedanken machen müssen, wo wir überall die Kontrolle wieder abgeben wollen.

2.1.5 Interprozesskommunikation und Synchronisation

Neben der Generierung von Threads bzw. Prozessen ist auch die Kommunikation zwischen diesen wichtig. Sie geschieht meist über geteilten Speicher bzw. Variablen.

Wir betrachten folgendes Beispiel in Pseudocode:

```
int i = 0;

par
    { i = i + 1; }
    { i = i * 2; }
end par;

print(i);
```

Nebenläufigkeit macht Programme nicht-deterministisch, d.h. es können je nach Scheduling unterschiedliche Ergebnisse herauskommen. So kann obiges Programm die Ausgaben 1, 2 oder sogar 0 erzeugen.

Die Frage, die sich stellt, ist: Welche Aktionen werden wirklich atomar ausgeführt? Durch Übersetzung des Programms in Byte- oder Maschinencode können sich folgende Instruktionen ergeben:

1. `i = i + 1` → `LOAD i; INC; STORE i;`

2. `i = i * 2` → `LOAD i; SHIFTL; STORE i;`

Dann führt der folgende Schedule zur Ausgabe 0:

```
(2) LOAD i;

(1) LOAD i;
(1) INC;
(1) STORE i;

(2) SHIFTL;
(2) STORE i;
```

Wir benötigen also Synchronisation zur Gewährleistung der atomaren Ausführung bestimmter Codeabschnitte, welche nebenläufig auf gleichen Ressourcen arbeiten. Solche Codeabschnitte nennen wir *kritische Bereiche*.

2.1.6 Synchronisation mit Semaphoren

Ein bekanntes Konzept zur Synchronisation nebenläufiger Threads oder Prozesse geht auf *Dijkstra* aus dem Jahre 1968 zurück. Dijkstra entwickelte einen abstrakten Datentyp mit dem Ziel, die atomare (ununterbrochene) Ausführung bestimmter Programmabschnitte zu garantieren. Seine *Semaphore* stellen zwei atomare Operationen zur Verfügung:

```
p(s) {
    if
        (s >= 1)
    then
        s = s - 1;
    else
        trage ausführenden Thread in Warteliste zu s ein und suspendiere ihn;
        s = s - 1;
}

v(s) {
    s = s + 1;

    if
        Warteliste zu s nicht leer
    then
        wecke ersten Prozess der Warteliste wieder auf
}
```

Dabei steht `p(s)` für passieren oder *passeer*, `v(s)` steht für verlassen oder *verlaat*.

Nun können wir bei unserem obigen Programm die Ausgabe 0 wie folgt verhindern:

```
int i = 0;
Semaphore s = 1;

par
    { p(s); i = i + 1; v(s); }
    { p(s); i = i * 2; v(s); }
end par;
```

Der Initialwert der Semaphore bestimmt dabei die maximale Anzahl der Prozesse im kritischen Bereich. Meist finden wir hier den Wert 1, solche Semaphore nennen wir *binäre Semaphore*.

Eine andere Anwendung von Semaphoren ist das *Producer-Consumer-Problem*: n Producer erzeugen Waren, die von m Consumern verbraucht werden. Eine einfache Lösung für dieses Problem verwendet einen unbeschränkten Buffer:

```
Semaphore num = 0;
```

Code für den Producer:

```
while (true) {
    product = produce();
    push(product, buffer);
    v(num);
}
```

Code für den Consumer:

```
while (true) {
    p(num);
    prod = pull(buffer);
    consume(prod);
}
```

Was nun noch fehlt ist die Synchronisation auf `buffer`. Diese kann durch eine weitere Semaphore hinzugefügt werden:

```
Semaphore num = 0;
Semaphore bufferAccess = 1;
```

Code für den Producer:

```
while (true) {
    product = produce();
    p(bufferAccess);
    push(product, buffer);
    v(bufferAccess);
    v(num);
}
```

Code für den Consumer:

```
while (true) {
    p(num);
    p(bufferAccess);
    prod = pull(buffer);
    v(bufferAccess);
    consume(prod);
}
```

Die Semaphore bringen jedoch auch einige Nachteile mit sich. Der Code mit Semaphoren wirkt schnell unstrukturiert und unübersichtlich. Außerdem können wir Semaphore nicht kompositionell verweden: So kann der einfache Code `p(s); p(s);` auf einer binären Semaphore `s` bereits einen *Deadlock* erzeugen.

Eine Verbesserung bieten hier die *Monitore*, die wir bereits aus der Vorlesung „Systemorientierte Informatik III: Betriebssysteme“ kennen. In der Tat verwendet Java einen Mechanismus ähnlich dieser Monitore zur Synchronisierung.

2.1.7 Dining Philosophers

Das Problem *dining philosophers* mit n Philosophen lässt sich wie folgt mit Hilfe von Semaphoren modellieren:

```
Semaphore[n] sticks = [1, 1, ..., 1];
```

Code für Philosoph i:

```
while (true) {
    think();

    p(stick[(i - 1) mod n]);
    p(stick[i]);

    eat();
```

```
    v(stick[(i - 1) mod n]);
    v(stick[i]);
}
```

Dabei tritt jedoch ein Deadlock auf, falls alle Philosophen gleichzeitig ihr linkes Stäbchen nehmen. Dieses Deadlock können wir durch Zurücklegen vermeiden:

```
while (true) {
    think();

    p(stick[(i - 1) mod n]);

    if
        (l(stick[i]) == 0)
    then
        v(stick[(i - 1) mod n]);
        continue;
    else
        p(stick[i]);

    eat();

    v(stick[(i - 1) mod n]);
    v(stick[i]);
}
```

Hier bezeichnet `l(s)` eine Lookup-Funktion, die uns den Wert einer Semaphore `s` zurückgibt.

Das Programm hat nun noch einen Livelock, d.h. einzelne Philosophen können verhungern. Diesen möchten wir hier nicht weiter behandeln.

2.2 Threads in Java

2.2.1 Die Klasse `Thread`

Die API von Java bietet im Package `java.lang` eine Klasse `Thread` an. Eigene Threads können von dieser abgeleitet werden. Der Code, der dann nebenläufig ausgeführt werden soll, wird in die Methode `run()` geschrieben. Nachdem wir einen neuen Thread einfach mit Hilfe seines Konstruktors erzeugt haben, können wir ihn zur nebenläufigen Ausführung mit der Methode `start()` starten.

Wir betrachten folgenden einfachen Thread:

```
public class ConcurrentPrint extends Thread {
    private String s;

    public ConcurrentPrint(String s) {
        this.s = s;
    }

    public void run() {
        while (true) {
            System.out.print(s + " ");
        }
    }

    public static void main(String[] args) {
        new ConcurrentPrint("a").start();
        new ConcurrentPrint("b").start();
    }
}
```

Starten des obigen Programms kann zu vielen möglichen Ausgaben führen:

```
a a b b a a b b ...
a a a b b ...
a b a a a b a a b b ...
a a a a a a a a a ...
```

Letztere ist dann garantiert, wenn kooperatives Scheduling vorliegt.

2.2.2 Das Interface `Runnable`

Java bietet keine Mehrfachvererbung. Deshalb ist eine Erweiterung der Klasse `Thread` häufig ungünstig. Eine Alternative bietet das Interface `Runnable`:

```
public class ConcurrentPrint implements Runnable {
    private String s;

    public ConcurrentPrint(String s) {
        this.s = s;
    }

    public void run() {
```

```
        while (true) {
            System.out.print(s + " ");
        }
    }

    public static void main(String[] args) {
        Runnable aThread = new ConcurrentPrint("a");
        Runnable bThread = new ConcurrentPrint("b");

        new Thread(aThread).start();
        new Thread(bThread).start();
    }
}
```

Beachte: Innerhalb der obigen Implementierung von `ConcurrentPrint` liefert `this` kein Objekt vom Typ `Thread` mehr. Das aktuelle Thread-Objekt erreicht man dann über die statische Methode `Thread.currentThread()`.

2.2.3 Eigenschaften von Thread-Objekten

Jedes Thread-Objekt in Java hat eine Reihe von Eigenschaften:

- *Name.* Beispiele für Namen von Threads sind „main-Thread“, „Thread-0“ oder „Thread-1“. Zugriff auf den Namen eines Threads erfolgt über die Methoden `getName()` und `setName(String)`. Man verwendet sie in der Regel zum Debuggen.
- *Zustand.* Jeder Thread befindet stets in einem bestimmten Zustand. Eine Übersicht dieser Zustände und der Zustandsübergänge folgt.

  Ein Thread-Objekt bleibt auch im Zustand *terminiert* noch solange erhalten, bis alle Referenzen auf ihn verworfen wurden.
- *Dämon.* Ein Thread kann mit Hilfe des Aufrufs `setDemon(true)` *vor* Aufruf der `start()`-Methode als Hintergrundthread deklariert werden. Die JVM terminiert, sobald nur noch Dämonenthreads laufen. Beispiele für solche Threads sind AWT-Threads oder der Garbage Collector.
- *Priorität.* Jeder Thread in Java hat eine bestimmte Priorität. Die genaue Staffelung ist plattformspezifisch.
- *Threadgruppen.* Threads können zur gleichzeitigen Behandlung auch in Gruppen eingeteilt werden.
- *Methode `sleep(long)`.* Lässt den Thread die angegebene Zeit schlafen. Kann eine `InterruptedException` werfen, welche aufgefangen werden muss.

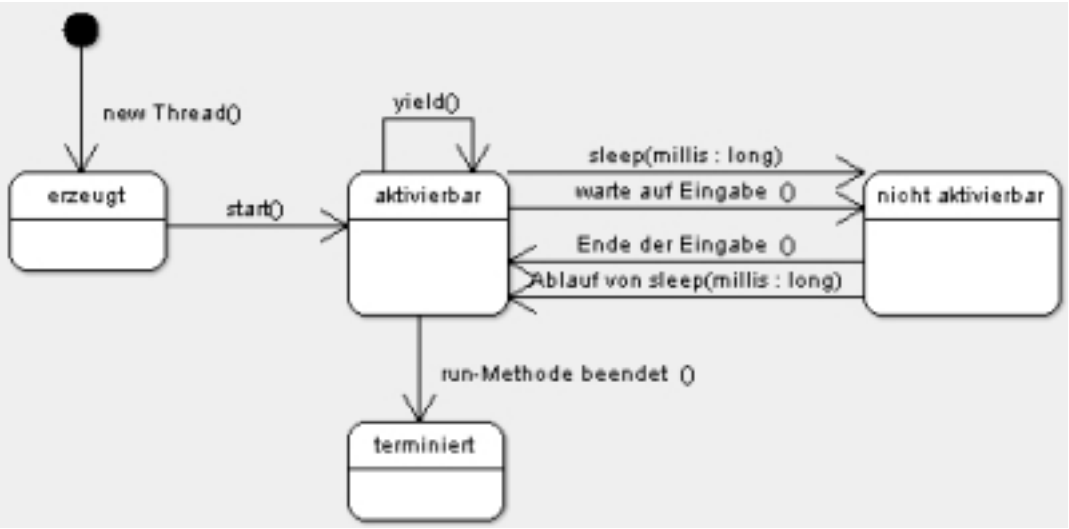

Abbildung 2.1: Zustände von Threads

2.2.4 Synchronisation von Threads

Zur Synchronisation von Threads bietet Java ein Monitor-ähnliches Konzept, das es erlaubt, Locks auf Objekten zu nehmen und wieder freizugeben.

Die Methoden eines Threads können in Java als `synchronized` deklariert werden. In allen synchronisierten Methoden eines Objektes darf sich dann maximal ein Thread zur Zeit befinden. Hierzu zählen auch Berechnungen, die in einer synchronisierten Methode aufgerufen werden, auch unsynchronisierte Methoden des gleichen Objektes. Dabei wird eine synchronisierte Methode nicht durch einen Aufruf von `sleep(long)` oder `yield()` verlassen.

Ferner besitzt jedes Objekt ein eigenes *Lock* . Beim Versuch der Ausführung einer Methode, die als `synchronized` deklariert ist, unterscheiden wir drei Fälle:

1. Ist das Lock freigegeben, so nimmt der Thread es sich.
2. Besitzt der Thread das Lock bereits, so passiert er.
3. Sonst wird der Thread suspendiert.

Das Lock wird wieder freigegeben, falls die Methode verlassen wird, in der es genommen wurde.

Im Vergleich zu Semaphoren wirkt der Ansatz von Java strukturierter, man kann kein unlock vergessen. Dennoch ist er weniger flexibel.

2.2.5 Die Beispielklasse `Account`

Ein einfaches Beispiel soll die Verwendung von `synchronized`- Methoden veranschaulichen. Wir betrachten eine Implementierung einer Klasse für ein Bankkonto:

```
public class Account {
    private double balance;

    public Account(double initialDeposit) {
        balance = initialDeposit;
    }

    public synchronized double getBalance() {
        return balance;
    }

    public synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

Wir möchten die Klasse nun wie folgt verwenden:

```
Account a = new Account(300);

...

a.deposit(100); // nebenläufig, erster Thread
a.deposit(100); // nebenläufig, zweiter Thread

...

System.out.println(a.getBalance());
```

Die Aufrufe der Methode `deposit(double)` sollen dabei nebenläufig von verschiedenen Threads aus erfolgen. Ohne das Schlüsselwort `synchronized` wäre die Ausgabe `500`, aber auch die Ausgabe `400` denkbar (vgl. dazu Abschnitt 2.1.4. Interprozesskommunikation und Synchronisation). Mit Anwendung des Schlüsselwortes ist eine Ausgabe von `500` garantiert.

2.2.6 Genauere Betrachtung von `synchronized`

Vererbte synchronisierte Methoden müssen nicht zwingend wieder synchronisiert sein. Wenn man solche Methoden überschreibt, so kann man das Schlüsselwort `synchronized` auch weglassen. Dies bezeichnet man als *verfeinerte Implementierung*. Die Methode der Oberklasse bleibt dabei `synchronized`. Andererseits können unsynchronisierte Methoden auch durch synchronisierte überschrieben werden.

Klassenmethoden, die als synchronisiert deklariert werden (`static synchronized`) haben keine Wechselwirkung mit synchronisierten Objektmethoden. Die Klasse hat also ein eigenes Lock.

Man kann sogar auf einzelne Anweisungen synchronisieren:

```
synchronized (expr) block
```

Dabei muss `expr` zu einem Objekt auswerten, dessen Lock dann zur Synchronisation verwendet wird. Streng genommen sind synchronisierte Methoden also nur syntaktischer Zucker: So steht die Methodendeklaration

```
synchronized A m(args) block
```

eigentlich für

```
A m(args) {
    synchronized (this) block
}
```

Einzelne Anweisungen zu synchronisieren ist sinnvoll, um weniger Code synchronisieren bzw. sequentialisieren zu müssen:

```
private double state;

public void calc() {
    double res;

    // do some really expensive computation
    ...

    // save the result to an instance variable
    synchronized (this) {
        state = res;
    }
}
```

Synchronisierung auf einzelne Anweisungen ist auch nützlich, um auf andere Objekte zu synchronisienen. Wir betrachten als Beispiel eine einfache Implementierung einer synchronisierten Collection:

```
class Store {
    public synchronized boolean hasSpace() {
        ...
    }

    public synchronized void insert(int i)
        throws NoSpaceAvailableException {

        ...
    }
}
```

Wir möchten diese Collection nun verwenden wie folgt:

```
Store s = new Store();

...

if (s.hasSpace()) {
    s.insert(42);
}
```

Dies führt jedoch zu Problemen, da wir nicht ausschließen können, dass zwischen den Aufrufen von `hasSpace()` und `insert(int)` ein Re-Schedule geschieht. Da sich das definieren spezieller Methoden für solche Fälle oft als unpraktikabel herausstellt, verwenden wir die obige Collection also besser folgendermaßen:

```
synchronized(s) {
    if (s.hasSpace()) {
        s.insert(42);
    }
}
```

2.2.7 Unterscheidung der Synchronisation im OO-Kontext

Wir bezeichnen synchronisierte Methoden und synchronisierte Anweisungen in Objektmethoden als *server-side synchronisation*. Synchronisation der Aufrufe eines Objektes bezeichnen wir als *client-side synchronisation*.

Aus Effizenzgründen werden Objekte der Java-API, insbesondere Collections, nicht mehr synchronisiert. Für Collections stehen aber synchronisierte Versionen über Wrapper wie `synchronizedCollection`, `synchronizedSet`, `synchronizedSortedSet`, `synchronizedList`, `synchronizedMap` oder `synchronizedSortedMap` zur Verfügung.

Sicheres Kopieren einer Liste in ein Array kann nun also auf zwei verschiedene Weisen bewerkstelligt werden: Als erstes legen wir eine Instanz einer synchronisierten Liste an:

```
List<Integer> unsyncList = new List<Integer>();

// fill the list
...

List<Integer> list = Collections.synchronizedList(unsyncList);
```

Nun können wir diese Liste entweder mit der einfachen Zeile

```
Integer[] a = list.toArray(new Integer[0]);
```

oder über

```
Integer[] b;

synchronized (list) {
    b = new Integer[list.size()];
    list.toArray(b);
}
```

in ein Array kopieren. Bei der zweiten, zweizeiligen Variante ist die Synchronisierung auf die Liste unabdingbar: Wir greifen in beiden Zeilen auf die Collection zu, und wir können nicht garantieren, dass nicht ein anderer Thread die Collection zwischenzeitig verändert. Dies ist ein klassisches Beispiel für client-side synchronisation.

2.2.8 Kommunikation zwischen Threads

Threads kommunizieren über geteilte Objekte miteinander. Wie finden wir nun heraus, wann eine Variable einen Wert enthält? Dafür gibt es mehrere Lösungsmöglichkeiten.

Die erste Möglichkeit ist die Anzeige des Veränderns einer Komponente des Objektes, beispielsweise durch Setzen eines booleschen Flags. Dies bringt

jedoch den Nachteil mit sich, dass das Prüfen auf das Boolesche Flag zu busy waiting führt. Deshalb suspendiert man mittels einer Methode des Objektes `wait()`, und weckt es mittels `notify()` oder `notifyAll()` wieder auf.

Das sieht zum Beispiel so aus:

```
public class C {
    private int state = 0;

    public synchronized void printMyState()
        throws InterruptedException {

        wait();
        System.out.println(state);
    }

    public synchronized void setValue(int v) {
        state = v;
        notify();
        System.out.println("value set");
    }
}
```

Zwei Threads führen nun die Methodenaufrufe `printNewState()` und `setValue(42)` nebenläufig aus. Nun ist die *einzig* mögliche Ausgabe

```
value set
42
```

Falls der Aufruf von `wait()` erst kommt, nachdem die Methode `setValue(int)` vom ersten Thread schon verlassen wurde, so führt dies nur zur Ausgabe `value set`.

Die Methoden `wait()`, `notify()` und `notifyAll()` haben dabei in Java folgende Semantik:

- `wait()` legt den ausführenden Thread schlafen und gibt das Lock des Objektes wieder frei.
- `notify()` erweckt einen schlafenden Thread des Objekts und führt mit eigener Berechnung fort. Der erweckte Thread bewirbt sich nun um das Lock. Wenn kein Thread schläft, dann geht das `notify()` verloren.
- `notifyAll()` tut das gleiche wie `notify()`, nur für alle Threads.

Wir möchten nun ein Programm schreiben, das alle Veränderungen des Zustands ausgibt:

```
...

private boolean modified = false; // zur Anzeige der Zustandsänderung

...

public synchronized void printNewState() {
    while (true) {
        if (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
    }
}

public synchronized void setValue(int v) {
    state = v;
    notify();
    modified = true;
    System.out.println("value set");
}
```

Ein Thread führt nun `printNewState()` aus, andere Threads verändern den Zustand mittels `setValue(int)`. Dies führt zu einem Problem: Bei mehreren setzenden Threads können einzelne Zwischenzustände verloren gehen. Also muss auch `setValue(int)` ggf. warten und wieder aufgeweckt werden:

```
public synchronized void printNewState() {
    while (true) {
        if (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
        notify();
    }
}
```

```
public synchronized void setValue(int v) {
    if (modified) {
        wait();
    }

    state = v;
    notify();
    modified = true;
    System.out.println("value set");
}
```

Nun ist es aber nicht gewährleistet, dass der Aufruf von `notify()` in `setValue(int)` den `printNewState`-Thread aufweckt! In Java lösen wir dieses Probem mit Hilfe von `notifyAll()` und nehmen dabei ein wenig busy waiting in Kauf:

```
public synchronized void printNewState() {
    while (true) {
        while (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
        notify();
    }
}

public synchronized void setValue(int v) {
    while (modified) {
        wait();
    }

    state = v;
    notifyAll();
    modified = true;
    System.out.println("value set");
}
```

Die Methode `wait()` ist in Java außerdem mehrmals überladen:

- `wait(long)` unterbricht die Ausführung für die angegebene Anzahl an Millisekunden.

- `wait(long, int)` unterbricht die Ausführung für die angegebende Anzahl an Milli- und Nanosekunden.

Anmerkung: Es ist dringend davon abzuraten, die Korrektheit des Programms auf diese Überladungen zu stützen!

Die Aufrufe `wait(0)`, `wait(0, 0)` und `wait()` führen alle dazu, dass der Thread solange wartet, bis er wieder aufgeweckt wird.

2.2.9 Fallstudie: Einelementiger Puffer

Ein einelementiger Puffer ist günstig zur Kommunikation zwischen Threads. Der Puffer kann leer oder voll sein. In einen leeren Puffer kann über eine Methode `put` ein Wert geschrieben werden, aus einem vollen Puffer kann mittels `take` der Wert entfernt werden. `take` suspendiert auf leerem Puffer, `put` suspendiert auf vollem Puffer.

```
public class Buffer1<T> {
    private T content;
    private boolean empty;


    public Buffer1() {
        empty = true;
    }

    public Buffer1(T content) {
        this.content = content;
        empty = false;
    }


    public synchronized T take() throws InterruptedException {
        while (empty) {
            wait();
        }

        empty = true;
        notifyAll();

        return content;
    }

    public synchronized void put(T o) throws InterruptedException {
        while (!empty) {
```

```
            wait();
        }

        empty = false;
        notifyAll();
        content = o;
    }

    public synchronized boolean isEmpty() {
        return empty;
    }
}
```

Unschön an der obigen Lösung ist, dass zuviele Threads erweckt werden. Können wir Threads auch gezielt erwecken? Ja! Dazu verwenden wir spezielle Objekte zur Synchronisation der taker und putter.

```
public class Buffer1<T> {
    private T content;
    private boolean empty;
    private Object r = new Object();
    private Object w = new Object();


    public Buffer1() {
        empty = true;
    }

    public Buffer1(T content) {
        this.content = content;
        empty = false;
    }


    public T take() throws InterruptedException {
        synchronized (r) {
            while (empty) {
                r.wait();
            }

            synchronized (w) {
                empty = true;
                w.notify();
```

```
                return content;
            }
        }
    }

    public void put(T o) throws InterruptedException {
        synchronized(w) {
            while (!empty) {
                w.wait();
            }

            synchronized (r) {
                empty = false;
                r.notify();
                content = o;
            }
        }
    }

    public boolean isEmpty() {
                return empty;
    }
}
```

Hier ist das `while` wichtig! Ein anderer Thread, der die Methode von außen betritt, könnte sonst einen wartenden (und gerade aufgeweckten) Thread ansonsten noch überholen!

2.2.10 Beenden von Threads

Java bietet mehrere Möglichkeiten, Threads zu beenden:

1. Beenden der `run()`-Methode
2. Abbruch der `run()`-Methode
3. Aufruf der `destroy()`-Methode (deprecated, z. T. nicht mehr implementiert)
4. Dämonthread und Programmende

Bei 1. und 2. werden alle Locks freigegeben. Bei 3. werden Locks nicht freigegeben, was diese Methode unkontrollierbar macht. Bei 4. sind die Locks egal.

Java sieht zusätzlich eine Möglichkeit zum Unterbrechen von Threads über *Interrupts* vor. Jeder Thread hat ein Flag, welches Interrupts anzeigt.

Die Methode `interrupt()` sendet einen Interrupt an einen Thread, das Flag wird gesetzt. Falls der Thread aufgrund eines Aufrufs von `sleep()` oder `wait()` schläft, wird er erweckt und eine `InterruptedException` geworfen.

```
synchronized (o) {
    ...

    try {
        ...

        o.wait();

        ...
    } catch (InterruptedException e) {
        ...
    }
}
```

Bei Interrupt nach dem Aufruf von `wait()` wird der `catch`-Block erst betreten, wenn der Thread das Lock auf das Objekt `o` des umschließenden `synchronized`-Blocks wieder erlangt hat!

Im Gegensatz dazu wird bei der Suspension durch `synchronized` der Thread nicht erweckt, sondern nur das Flag gesetzt.

Die Methode `public boolean isInterrupted()` testet, ob ein Thread Interrupts erhalten hat. `public static boolean interrupted()` testet den aktuellen Thread auf Interrupt und löscht das Interrupted-Flag.

Falls man also in einer `synchronized`-Methode auf Interrupts reagieren möchte, ist dies wie folgt möglich:

```
synchronized void m(...) {
    ...

    if (Thread.currentThread().isInterrupted()) {
        throw new InterruptedException();
    }
}
```

Falls eine `InterruptedException` aufgefangen wird, wird das Flag ebenfalls gelöscht. Dann muss man das Flag erneut setzten!

2.3 Verteilte Programmierung in Java

Java bietet als Abstraktion der Netzwerkkommunikation die *Remote Method Invocation (RMI)* an. Hiermit können Remote-Objekte auf anderen Rechnern verwendet werden, als wären es lokale Objekte. Argumente und Ergebnisse müssen hierbei in Byte-Folgen umgewandelt werden.

2.3.1 Serialisierung von Daten

Die Serialisierung eines Objektes o liefert eine Byte-Sequenz, die Deserialisierung der Byte-Sequenz liefert ein neues Objekt o'. Beide Objekte sollen bezüglich ihres Verhaltens gleich sein, haben aber unterschiedliche Objektidentitäten.

Die (De-)Serialisierung erfolgt rekursiv. Enthaltene Objekte müssen also auch (de-)serialisiert werden. Dafür bietet Java das Interface `Serializable`:

```
public class C implements java.io.Serializable { ...  }
```

Es stellt Methoden zum (De-)Serialisieren zur Verfügung. Hierbei können bestimmte zeit- oder sicherheitskritische Teile eines Objektes mit Hilfe des Schlüsselwortes `transient` ausgeblendet werden:

```
protected transient String password;
```

Transiente Werte sollten nach dem Deserialisieren explizit gesetzt (z.B. Timer) bzw. nicht verwendet werden (z.B. Passwort).

Zum Serialisieren verwendet man die Klasse `ObjectOutputStream`, zum Deserialisieren die Klasse `ObjectInputStream`. Deren Konstruktoren wird ein `OutputStream` bzw. `InputStream` übergeben.

Danach können Objekte mittels `public void writeObject(Object o)` geschrieben und mit Hilfe von `public final Object readObject()` und Cast in den entsprechenden Typ gelesen werden.

Oder man umgeht diese expliziten Aufrufe; dies wird in Java durch Remote Method Invocation ermöglicht.

2.3.2 Remote Method Invocation (RMI)

In der OO-Programmierung haben wir eine Client-Server-Sicht von Objekten. Beim Aufruf einer Methode wird das aufrufende Objekt als Klient und das aufgerufene Objekt als Server gesehen.

Im verteilten Kontext werden Nachrichten dann echte Nachrichten im Internet (TCP). Prozesse, die miteinander kommunizieren, sind dann:

- Server, welche Informationen zur Verfügung stellen

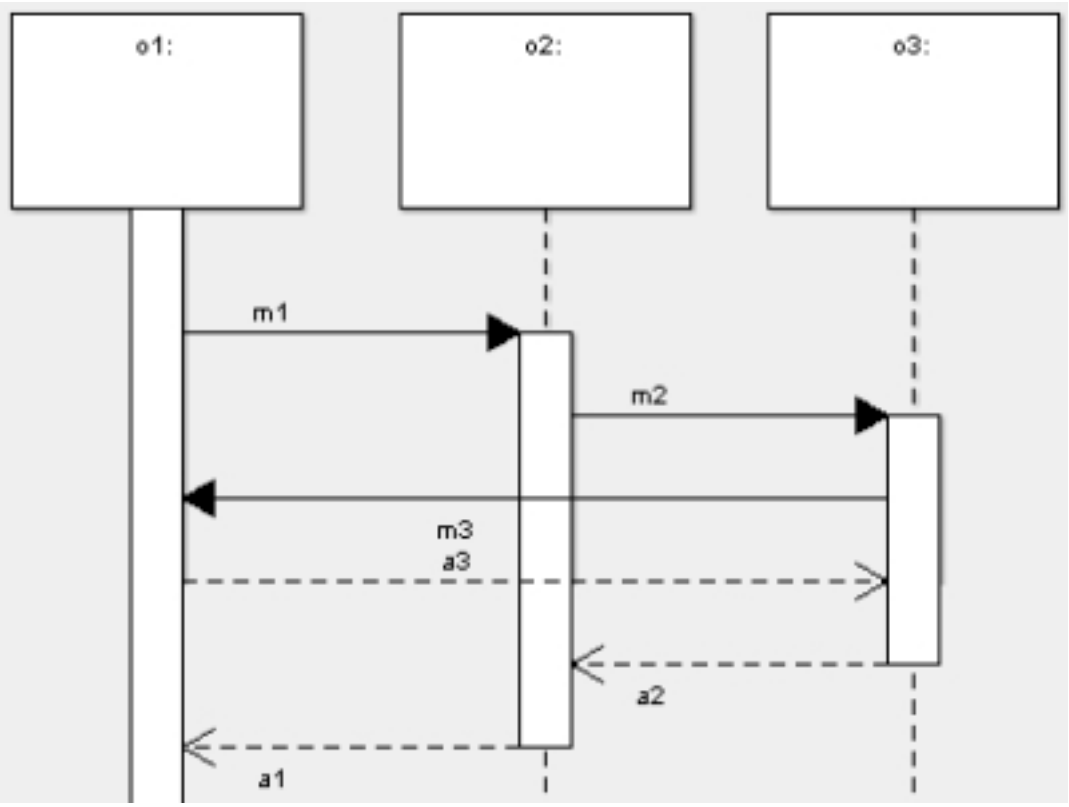

Abbildung 2.2: Remote Method Invocation in Java

- Klienten, die dies anfragen

Hiermit können beliebige Kommunikationsmuster (z.B. peer-to-peer) abgebildet werden. Die Idee von RMI geht auf *Remote Procedure Call (RPC)* zurück, welches für C entwickelt wurde.

Zunächst benötigt ein RMI-Client eine Referenz auf das Remote-Objekt. Dazu dient die RMI-Registrierung. Er fragt die Referenz mit Hilfe einer URL an:

```
rmi://hostname:port/servicename
```

Dabei kann `hostname` ein Rechnername oder eine IP-Adresse sein, `servicename` ist ein String, der ein Objekt beschreibt. Der Standardport von RMI ist 1099.

Es gibt auch noch eine zweite Möglichkeit, ein Remote-Objekt zu erhalten, und zwar als Argument eines Methodenaufrufs. In der Regel verwendet man die oben genannte Registrierung nur für den „Erstkontakt“, danach werden Objekte ausgetauscht und transparent (wie lokale Objekte) verwendet.

Solche Objekte können auf beliebigen Rechnern verteilt liegen. Methodenaufrufe bei entfernten Objekten werden durch Netzwerkkommunikation

umgesetzt.

Das Interface für RMI ist aufgeteilt in Stub und Skeleton. Seit Java 5 sind diese jedoch nicht mehr sichtbar.

Die Netzwerkkommunikation erfolgt über TCP/IP, aber auch diese ist für den Anwendungsprogrammierer nicht sichtbar.

Die Parameter und der Rückgabewert einer Methode müssen dafür natürlich in Bytes umgewandelt werden:

- Bei Remote-Objekten wird nur eine Referenz des Objektes übertragen.
- Serializable-Objekte werden in `byte[]` umgewandelt. Auf der anderen Seite wird dann eine Kopie des Objektes angelegt.
- Primitive Werte werden kopiert.

Um Objekte von entfernten Knoten verwenden zu können, muss zunächst ein RMI Service Interface definiert werden.

```
import java.rmi.*;

public interface FlipServer extends Remote {
    public void flip() throws RemoteException;
    public boolean getState() throws RemoteException;
}
```

Eine Implementierung des RMI-Interfaces sieht auf der Serverseite dann zum Beispiel so aus:

```
public class FlipServerImpl
    extends java.rmi.server.UnicastRemoteObject
    implements FlipServer {

    private boolean state;

    public FlipServerImpl() throws RemoteException {
        state = false;
    }

    public void flip() {
        state = !state;
    }

    public boolean getState() {
        return state;
```

```
    }
}
```

Früher musste man die Stub- und Skeleton-Klassen mit `rmic` generieren - das ist jetzt nicht mehr notwendig.

2.3.3 RMI-Registierung

Ein RMI-Registry-Server kann mittels `rmiregistry` unter UNIX gestartet werden. Seine Verwendung wird am folgenden Beispiel klar:

```
public class Server {
    public static void main(String[] args) {
        try {
            LocateRegistry.getRegistry();
            FlipServerImpl server = new FlipServerImpl();

            String registry;
            if (args.length >= 1) {
                registry = args[0];
            } else {
                registry = "localhost";
            }

            String url = "rmi://" + registry + "/FlipServer";
            Naming.rebind(url, server);
        } catch (RemoteException e) {
            ...
        } catch (MalformedURLException e) {
            ...
        }
    }
}

public class Client {
    public static void main(String[] args) {
        try {
            String registry;
            if (args.length >= 1) {
                registry = args[0];
            } else {
                registry = "localhost";
            }
```

```
            String url = "rmi://" + registry + "/FlipServer";
            FlipServerImpl s = (FlipServerImpl)Naming.lookup(url);

            s.flip();
            System.out.println("State: " + s.getState());
        } catch (MalformedURLException e) {
            ...
        } catch (RemoteException e) {
            ...
        } catch (NotBoundException e) {
            ...
        }
    }
}
```

Beachte: RMI stellt eine Fortsetzung der sequentiellen Programmierung auf verteilte Objekte dar. Dennoch können mehrere verteilte Prozesse auf ein Objekt „gleichzeitig“ zugreifen. Man muss also auch hier das Problem der nebenläufigen Synchronisation beachten!

Dynamisches Laden, Sicherheitskonzepte oder Verteilte Garbage Collection sind weitere Aspekte von Java RMI.

Kapitel 3

Funktionale Programmierung

Die funktionale Programmierung mit Haskell '98 bietet eine Reihe von Vorteilen:

- hohes Abstraktionsniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte, deshalb leichtere Codeoptimierung und bessere Verständlichkeit
- Programmierung über Eigenschaften, nicht über zeitlichen Ablauf
- implizite Speicherverwaltung
- einfachere Korrektheitsbeweise, Verifikation
- kompakte Source-Code-Größe, deshalb kürzere Entwicklungszeit, lesbarere Programme, bessere Wartbarkeit
- modularer Programmaufbau, Polymorphismus, Funktionen höherer Ordnung, Wiederverwendbarkeit von Code

In funktionalen Programmen stellt eine *Variable* einen unbekannten Wert dar. Ein *Programm* ist Menge von Funktionsdefinitionen. Der Speicher ist nicht explizit verwendbar, sondern wird automatisch verwaltet und aufgeräumt. Ein *Programmablauf* besteht aus der Reduktion von Ausdrücken. Dies geht auf die mathematische Theorie des λ-Kalküls, Church '41, zurück.

3.1 Funktions- und Typdefinitionen

Da in der Mathematik eine Variable für unbekannte (beliebige) Werte steht, arbeiten wir dort oft mit Ausdrücken wie

$x^2 - 4x + 4 = 0 \Leftrightarrow x = 2$

In imperativen Sprachen sehen wir hingegen häufig Ausdrücke wie

$x = x + 1$

welche einen Widerspruch zur Mathematik darstellen. Variablen als unbekannte Werte finden wir aber auch in der funktionalen Programmierung!

Während Funktionen in der Mathematik zur Berechnung dienen, verwenden wir Prozeduren oder Funktionen in Programmiersprachen zur Strukturierung. Dort ergibt sich aber wegen Seiteneffekten kein wirklicher Zusammenhang. In funktionalen Programmiersprachen gibt es jedoch keine Seiteneffekte, somit liefert jeder Funktionsaufruf mit gleichen Argumenten das gleiche Ergebnis.

Funktionsdefinitionen sehen in Haskell folgendermaßen aus:

```
f x1 ... xn = e
```

Dabei ist `f` der Funktionsname, `x1` bis `xn` sind formale Parameter bzw. Variablen, und `e` ist der Rumpf, ein Ausdruck über `x1` bis `xn`.

In Haskell sind nun folgende Ausdrücke möglich:

1. Zahlen: (`3`, `3.14159`)
2. Basisoperationen: (`3 + 4`, `5 * 7`)
3. Funktionsanwendungen: `(f e1 ... en)`. Die Außenklammerung kann entfallen, falls der Zusammenhang klar ist.
4. Bedingte Ausdrücke: `(if b then e1 else e2)`

Nehmen wir an, die Quadratfunktion

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

sei in einer Datei `square.hs` gespeichert. Dann kann man die Interpreter `hugs` oder `ghci`, welche sich im Verzeichnis ∼`haskell/bin/` befinden, verwenden wie folgt:

```
~haskell/bin/

> ghci
GHCi, version 6.10.3: http://www.haskell.org/ghc/   :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l square
[1 of 1] Compiling Main                ( square.hs, interpreted )
```

```
Ok, modules loaded: Main.
*Main> square 3
9
*Main> square (3 + 1)
16
*Main> :q
Leaving GHCi.
```

Eine Funktion zur Berechnung des Minimums zweier Zahlen kann in Haskell so aussehen:

```
min x y = if x <= y then x else y
```

Als nächstes betrachten wir die Fakultätsfunktion. Diese ist mathematisch wie folgt definiert:

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)!, & \text{sonst} \end{cases}$$

In Haskell setzen wir diese Funktion so um:

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

3.1.1 Auswertung

Die Auswertung von Funktionsdefinitionen in Haskell erfolgt durch orientierte Berechnung von links nach rechts: Zuerst werden die aktuellen Parameter gebunden, also die formalen Parameter durch die aktuellen ersetzt. Dann wird die linke durch die rechte Seite ersetzt.

Abbildung 3.1 zeigt, auf welche Art und Weise Funktionen ausgewertet werden können. Der rechte Ast zeigt, wie eine Funktion in Java auswertet wird, der linke ähnelt der Auswertung in Haskell, wenn man doppelte Berechnungen wie die von `3 + 1` weglässt.

Als weiteres Beispiel folgt die Auswertung eines Aufrufs unserer Funktion `fac`:

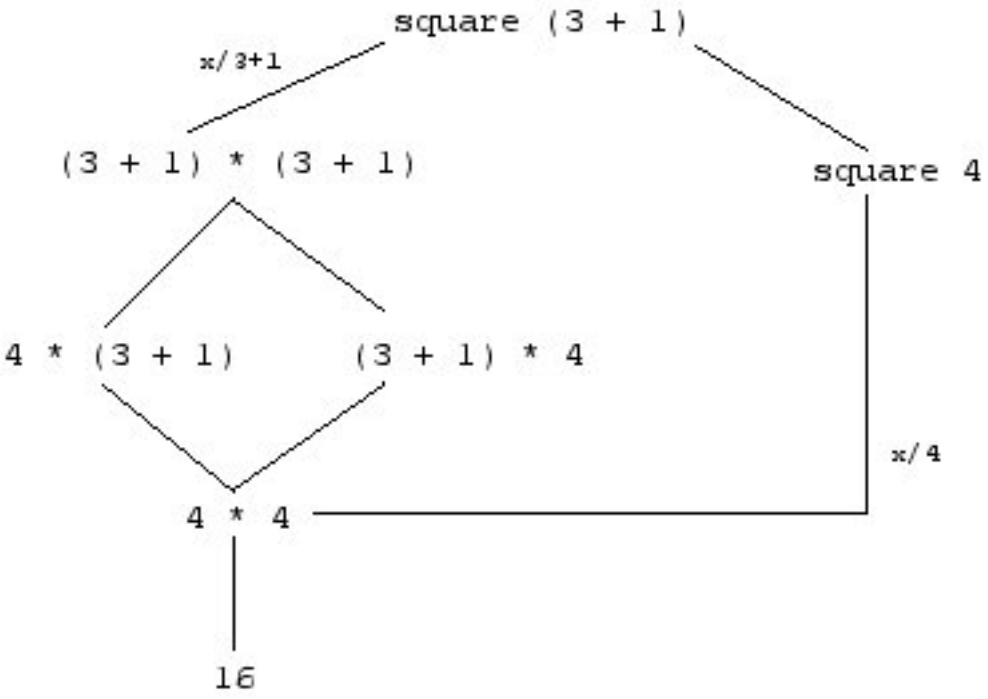

Abbildung 3.1: Mögliche Auswertungen von Funktionen

```
fac 2 = if 2 == 0 then 1 else 2 * fac (2 - 1)                    (3.1)
      = if false then 1 else 2 * fac (2 - 1)                     (3.2)
      = 2 * fac (2 - 1)                                          (3.3)
      = 2 * fac 1                                                (3.4)
      = 2 * (if 1 == 0 then 1 else 1 * fac (1 - 1))              (3.5)
      = 2 * (if false then 1 else 1 * fac (1 - 1))               (3.6)
      = 2 * 1 * fac (1 - 1)                                      (3.7)
      = 2 * 1 * fac 0                                            (3.8)
      = 2 * 1 * (if 0 == 0 then 1 else 0 * fac (0 - 1))          (3.9)
      = 2 * 1 * (if true then 1 else 0 * fac (0 - 1))            (3.10)
      = 2 * 1 * 1                                                (3.11)
      = 2 * 1                                                    (3.12)
      = 2                                                        (3.13)
```

Wir möchten nun eine effiziente Funktion zur Berechnung von Fibonacci-Zahlen entwickeln. Unsere erste Variante ist direkt durch die mathematische

Definition motiviert:

```
fib1 n = if n == 0
    then 0
    else if n == 1
        then 1
        else fib1 (n - 1) + fib (n - 2)
```

Diese Variante ist aber äußerst ineffizient: Ihre Laufzeit liegt in in $O(2^n)$.

Wie können wir unsere erste Variante verbessern? Wir berechnen die Fibonacci-Zahlen von unten: Die Zahlen werden von 0 an aufgezählt, bis die n-te Zahl erreicht ist: 0 1 1 2 3 ... $fib(n)$.

Diese Programmiertechnik ist bekannt unter dem Namen *Akkumulatortechnik*. Wir werden nun also die beiden vorigen Zahlen stets als Parameter mitführen:

```
fib2' fibn fibnp1 n = if n == 0
    then fibn
    else fib2' fibnp1 (fibn + fibnp1) (n - 1)

fib2 n = fib2' 0 1 n
```

Dabei ist `fibn` ist die n-te Fibonacci-Zahl, `fibnp1` die (n+1)-te. Dadurch erreichen wir lineare Laufzeit.

Aus softwaretechnischer Sicht ist unsere zweite Variante aber unschön: Wir wollen natürlich andere Aufrufe von `fib2'` von außen vermeiden. Wie das funktioniert ist im nächsten Abschnitt beschrieben.

3.1.2 Lokale Definitionen

Haskell bietet mehrere Möglichkeiten, Funktionen lokal zu definieren. Eine Möglichkeit stellt das Schlüsselwort `where` dar:

```
fib2 n = fib2' 0 1 n
  where fib2' fibn fibnp1 n = if n == 0
          then fibn
          else fib2' fibnp1 (fibn + fibnp1) (n - 1)
```

`where`-Definitionen sind in der vorhergehenden Gleichung sichtbar, außerhalb sind sie unsichtbar.

Alternativ können wir auch das Schlüsselwort `let` verwenden:

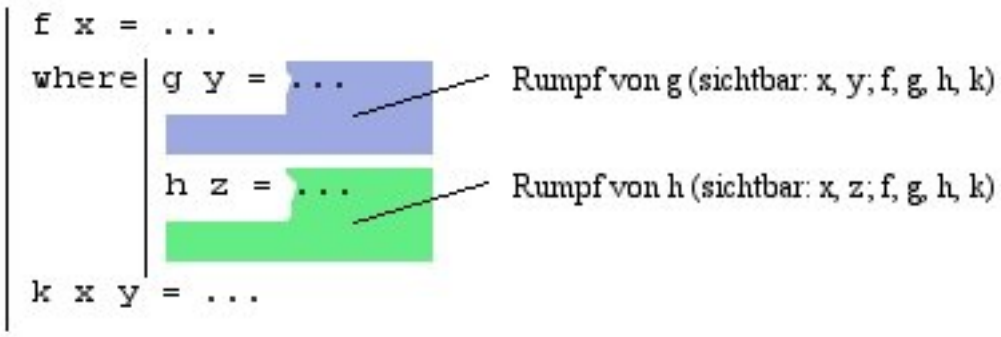

Abbildung 3.2: Layout-Regel in Haskell

```
fib2 n =
  let fib2' fibn fibnp1 n = if n == 0
        then fibn
        else fib2' fibnp1 (fibn + fibnp1) (n - 1)
  in fib2' 0 1 n
```

Dabei ist `let` im Gegensatz zu `where` ein *Ausdruck*. Das durch `let` definierte `fib2'` ist nur innerhalb des Abschnitts nach `in` sichtbar.

`let` ... `in` ... kann als beliebiger Ausdruck auftreten: So wird

```
(let x = 3
     y = 1
 in x + y) + 2
```

ausgewertet zu 6.

Die Syntax von Haskell verlangt bei der Definition solcher Blöcke keine Klammerung. In Haskell gilt die *Layout-Regel (off-side rule)*: Das nächste Symbol hinter `where` oder `let`, das kein Whitespace ist, definiert einen *Block*:

- Beginnt nun die Folgezeile rechts vom Block, so gehört sie zur gleichen Definition.
- Beginnt die Folgezeile am Blockrand, so beginnt hier eine neue Definition im Block.
- Beginnt die Folgezeile aber links vom Block, so wird der Block davor hier beendet.

Lokale Definitionen bieten eine Reihe von Vorteilen:

- Namenskonflikte können vermieden werden
- falsche Benutzung von Hilfsfunktionen kann vermieden werden
- bessere Lesbarkeit
- Mehrfachberechnungen können vermieden werden
- weniger Parameter bei Hilfsfunktionen

Ein Beispiel zur Vermeidung von Mehrfachberechnungen: Statt der unübersichtlichen Funktionsdefinition

```
f x y = y * (1 - y) + (1 + x * y) * (1 - y) + x * y
```

schreiben wir besser

```
f x y = let a = 1 - y
            b = x * y
        in y * a + (1 + b) * a + b
```

Dass uns `let` und `where` hilft, Parameter bei Hilfsfunktionen zu sparen, soll folgendes Beispiel verdeutlichen:

Das Prädikat isPrim soll überprüfen, ob es sich bei der übergebenen Zahl um eine Primzahl handelt. In Haskell drücken wir das so aus:

```
isPrim n = n /= 1 && checkDiv(div n 2)
  where checkDiv m =
          m == 1 || mod n m /= 0 && checkDiv(m - 1)
```

Hier drückt `/=` Ungleichheit aus, `&&` steht für eine Konjunktion, `||` für das logische Oder und `div` für die gannzzahlige Division.

In der letzten Zeile brauchen wir keine weitere Klammerung, da `&&` stärker bindet als `||`. Das `n` ist in `checkDiv` sichtbar, weil letzteres lokal definiert ist.

3.2 Basisdatentypen

3.2.1 Basisdatentypen in Haskell

Ganze Zahlen

Einen Basisdatentypen von Haskell haben wir oben bereits verwendet: Die ganzen Zahlen.

`Int`: Werte -2^{31} ... $2^{31} - 1$
`Integer`: beliebig groß (nur durch Speicher begrenzt)

Operationen: `+ - * div mod`
Vergleiche: `< > <= >= == /=`

Boolesche Werte

Ein weiterer Basisdatentyp sind die booleschen Werte:

`Bool`: `True False`

Operationen: `&& || not == /=`

Hier steht `==` für äquivalent, `/=` steht für das ausschließende Oder (XOR).

Gleitkommazahlen

Gleitkommazahlen sind ebenfalls ein Basisdatentyp:

`Float`: `0.3 -1.5e-2`

Operationen: wie `Int`, aber `/` statt `div`, kein `mod`

Zeichen

Und auch ASCII-Zeichen sind ein Haskell-Basisdatentyp:

`Char`: `'a' '\n' '\NUL' '\214'`

Operationen: `chr :: Int -> Char`, `ord :: Char -> Int`

Mit `::` werden in Haskell optionale Typannotationen beschrieben.

3.2.2 Typannotationen

Alle Werte und Ausdrücke in Haskell haben einen Typ, welcher auch annotiert werden kann:

```
3 :: Int
```

In diesem Beispiel ist `3` ein Wert bzw. Ausdruck, und `Int` ist ein Typausdruck. Weitere Beispiele für Typannotationen sind:

```
3 :: Integer
(3 == 4) || True :: Bool
(3 == (4 :: Int)) || True :: Bool
```

Wir können auch Typannotationen für Funktionen angeben. Diese werden in eine seperate Zeile geschrieben:

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

Aber was ist der Typ von `min` (siehe Abschnitt 3.1)? Der sieht folgendermaßen aus:

```
min :: Int -> Int -> Int
```

Auch zwischen den Argumenttypen wird also ein Funktionspfeil geschrieben.

3.2.3 Algebraische Datenstrukturen

Eigene Datenstrukturen können als neue Datentypen definiert werden. Werte werden mittels *Konstruktoren* aufgebaut. Konstruktoren sind frei interpretierte Funktionen, und damit nicht reduzierbar.

Definition eines algebraischen Datentyps

Algebraische Datentypen werden in Haskell wie folgt definiert:

`data` $\tau = c_1\tau_{11}...\tau_{1n_1}$ | ... | $c_k\tau_{k1}...\tau_{kn_k}$

wobei

- τ der neu definierte Typ ist
- $c_1, ..., c_k$ die definierten Konstruktoren sind und
- $\tau_{i1}, ..., \tau_{in_i}$ die Argumenttypen des Konstruktors sind, also
  c_i :: τ_{i1} -> ... -> τ_{in_i} -> τ

Beachte: Sowohl Typen als auch Konstruktoren müssen in Haskell mit Großbuchstaben beginnen!

Beispiele

1. Aufzählungstyp (nur 0-stellige Konstruktoren):
   `data Color = Red | Blue | Yellow` definiert genau die drei Werte des Typs `Color`. `Bool` ist auch ein Aufzählungstyp.

2. Verbundtyp (nur ein Konstruktor):

   ```
   data Complex = Complex Float Float
   Complex 3.0 4.0 :: Complex
   ```

   Dies ist erlaubt, da Haskell mit getrennten Namensräumen für Typen und Konstruktoren arbeitet.

   Wie selektieren wir nun einzelne Komponenten? In Haskell verwenden wir *pattern matching* statt expliziter Selektionsfunktionen:

   ```
   addC :: Complex -> Complex -> Complex
   addC (Complex r1 i1) (Complex r2 i2) =
       Complex (r1 + r2) (i1 + i2)
   ```

3. Listen (zunächst von `Int`s):
   `data List = Nil | Cons Int List`

   Hier steht `Nil` für die leere Liste. Die Funktion `append` erlaubt das Zusammenhängen von Listen:

   ```
   append :: List -> List -> List
   append Nil ys = ys
   append (Cons x xs) ys = Cons x (append xs ys)
   ```

   Dies ist eine Definition mit Hilfe mehrerer Gleichungen, wobei die erste passende gewählt wird. Ein Funktionsaufruf könnte zum Beispiel so reduziert werden:

   ```
     append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)
   = Cons 1 (append (Cons 2 Nil) (Cons 3 Nil))
   = Cons 1 (Cons 2 (append Nil (Cons 3 Nil)))
   = Cons 1 (Cons 2 (Cons 3 Nil))
   ```

In Haskell sind Listen vordefiniert mit:

`data [Int] = [] | Int:[Int]`

`[]` entspricht `Nil`, `:` entspricht `Cons` und `[Int]` entspricht `List`.

Dabei ist der Operator `:` rechtsassoziativ, also es gilt:

`1:(2:(3:[]))` entspricht `1:2:3:[]`

und der abkürzenden Schreibweise:

`[1,2,3]`

Außerdem bietet uns Haskell den Operator `++` statt `appends`:

```
[] ++ ys = ys
(x:xs) ++ ys = x : xs++ys
```

Operatoren sind zweistellige Funktionen, die infix geschrieben werden und mit Sonderzeichen beginnen. Durch Klammerung werden sie zu normalen Funktionen:

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

`[1] ++ [2]` entspricht `(++) [1] [2]`.

Umgekehrt können zweistellige Funktionen durch einfache Anführungszeichen `'...'` infix verwendet werden:

`div 4 2` entspricht `4 'div' 2`.

Für selbstdefinierte Datentypen besteht nicht automatisch die Möglichkeit, diese vergleichen oder ausgeben zu können. Hierzu kann man das Schlüüselwort `deriving` verwenden:

`data MyType = ... deriving (Eq, Show)`

3.3 Polymorphismus

Um den Polymorphismus in Haskell zu erläutern wollen wir als Beispiel die Länge einer Liste bestimmen:

```
length :: [Int] -> Int
length Nil = 0
length (_:xs) = 1 + length xs
```

Diese Definition funktioniert natürlich auch für andere Listen, beispielsweise vom Typ `[Char]`, `[Bool]` oder `[[Int]]`.

Allgemein könnte man also sagen:

`length :: ∀ Type` τ `: [`τ`] -> Int`

was in Haskell durch Typvariablen ausgedrückt wird:

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

Was ist der Typ von `(++)`?

```
(++) :: [a] -> [a] -> [a]
```

Es können also nur Listen mit gleichem Argumenttyp zusammengebaut werden. Wir betrachten ein weiteres Beispiel:

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

Dies funktioniert, da `[a]` ein Listentyp ist, und `[x]` eine einelementige Liste (entspricht `x:[]`). Die beiden Regeln dürfen wir aber nicht vertauschen, sonst terminiert kein Aufruf der Funktion!

Wie können wir nun selber polymorphe Datentypen definieren? Hierzu gibt es in Haskell Typkonstruktoren zum Aufbau von Typen:

`data` $\kappa\ a_1...a_m = c_1\tau_{11}...\tau_{1n_1}$ `|` `...` `|` $c_k\tau_{k1}...\tau_{kn_k}$

Diese Datentypdefinitionen sehen also ähnlich aus wie zuvor, aber hier ist

- κ Typkonstruktor (kein Typ)
- $a_1...a_m$ Typvariablen
- τ_{ik} Typen, welche Typvariablen sein oder verwenden können

Funktionen und Konstruktoren werden auf Werte bzw. Ausdrücke angewandt. Analog werden Typkonstruktoren auf Typen angewandt und erzeugen Typen.

Als Beispiel für einen polymorphen Datentyp möchten wir partielle Werte in Haskell modellieren:

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

Dann sind `Maybe Int` oder `Maybe (Maybe Int)` Typen.

Konstruktoren können auch auf Typvariablen angewandt werden. Es ergeben sich sog. polymorphe Typen.

```
isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing (Just _) = False
```

Ein weiteres gutes Beispiel für einen polymorphen Datentyp ist der Binärbaum:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

height :: Tree a -> Int
height (Leaf _) = 1
height (Node tl tr) = 1 + max (height tl) (height tr)
```

In Haskell sind auch polymorphe Listen als syntaktischer Zucker vordefiniert wie:

```
data [a] = [] | a:[a]
```

Diese Definition lässt sich in Haskell zwar so nicht direkt übersetzen, sie kann aber wie folgt verstanden werden: Die eckigen Klammern um `[a]` sind der Typkonstruktor für Listen, `a` ist der Elementtyp, `[]` ist die leere Liste und `:` ist der Listenkonstruktor.

In der Verwendung entsprechen sich die folgenden Ausdrücke:

```
(:) 1 ((:) 2 ((:) 3 []))
1 : (2 : (3 : []))
1 : 2 : 3 : []
[1,2,3]
```

Nach obiger Definition sind also Listentypen wie `[Int]`, `[Bool]` oder `[[Int]]` möglich.

Einige Funktionen auf Listen sind beispielsweise `head`, `tail`, `last`, `concat` und `!!`:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

last :: [a] -> a
last [x] = x
last (_:xs) = last xs

concat :: [[a]] -> [a]
concat [] = []
concat (l:ls) = l ++ concat ls

(!!) :: [a] -> Int -> a
(x:xs)!!n = if n = 0 then x
                     else xs!!(n - 1)
```

Für die letzte Funktion sind auch folgende Definitionen möglich:

```
(x:xs)!!0 = x
(x:xs)!!(n+1) = xs!!n
```

oder

```
(x:xs)!!0 = x
(x:xs)!!n = xs!!(n-1)
```

Zeichenketten sind in Haskell als Listen von Zeichen definiert:

```
type String = [Char]
```

So entspricht etwas die Zeichenkette `"Hallo"` der Liste `'H':'a':'l':'l':'o':[]`. Aus diesem Grund funktionieren alle Listenfunktionen auch für Strings: `length("Hallo"++Leute!")` wird ausgewertet zu `11`.

Weitere in Haskell vordefinierte Typkonstruktoren sind:

- Vereinigung zweier Typen

  ```
  data Either a b = Left a | Right b
  ```

Damit können zum Beispiel Werte „unterschiedlicher Typen“ in eine Liste geschrieben werden:

```
[Left 42, Right "Hallo"]::[Either Int String]
```

- Tupel

```
data (,) a b = (,) a b
data (,,) a b c = (,,) a b c
```

Auch hierauf sind bereits einige Funktionen definiert:

```
(3,True) :: (Int,Bool)

fst :: (a,b) -> a
fst (x, _) = x

snd :: (a,b) -> b
snd (_, y) = y

zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys


unzip :: [(a, b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):xys) =
    let (xs,ys) = unzip xys in
                (x:xs, y:ys)
```

Hier wird `zip (unzip l)` zwar zu `l` ausgewertet, aber `unzip (zip l1 l2)` nicht zu `(l1,l2)`!

3.4 Pattern Matching

Bisher kennen wir *pattern matching* aus Funktionsdefinitionen:

`f` $pat_{11}...pat_{1n}$ `=` l_1
`...`
`f` $pat_{k1}...pat_{kn}$ `=` l_k

Haskell wählt die erste Regel mit passender linker Seite und wendet diese an. Überlappende Regeln sind dabei oft unschön und sollten besser vermieden werden.

3.4.1 Aufbau der Pattern

- `x` *(Variable)*: passt immer, Variable wird an aktuellen Wert gebunden.
- `_` *(Wildcard)*: passt immer, keine Bindung.
- `c` $pat_1...pat_k$ wobei `c` k-stelliger Konstruktor: passt, falls gleicher Konstruktor und Argumente passen auf $pat_1, ..., pat_k$
- `x@pat` *(as pattern)*: passt, falls `pat` passt; zusätzlich wird `x` an gesamten Wert gebunden

  Damit können auch überlappende Pattern vermieden werden:

  ```
  last [x] = x
  last (x:xs@(_:_)) = last xs
  ```

- `(n+k)` wobei `k` ganze Zahl größer 0: passt auf alle Zahlen, die größer gleich `k` sind, wobei `n` an aktuellen Wert abzüglich `k` gebunden wird

  ```
  fac 0 = 1
  fac m@(n+1) = m * fac n
  ```

Pattern können auch bei `let` und `where` verwendet werden:

```
unzip ((x,y):xys) = (x:xs,y:ys)
    where (xs,ys) = unzip xys
```

3.4.2 Case-Ausdrücke

Manchmal ist es auch praktisch, mittels pattern matching in Ausdrücke zu verzweigen: So definiert

`case e of` pat_1 `->` e_1 `...` pat_n `->` e_n

einen Ausdruck mit Typ von $e_1, ..., e_n$, welche alle den gleichen Typ haben müssen. $e, pat_1, ..., pat_n$ müssen ebenfalls den gleichen Typ haben.

Das Ergebnis von `e` wird hier der Reihe nach gegen pat_1 bis pat_n gematcht. Falls ein pattern passt, wird das ganze case durch das zugehörige e_i ersetzt.

Als Beispiel betrachten wir das Extrahieren der Zeilen eines Strings:

```
lines :: String -> [String]
lines "" = []
lines ('\n':cs) = "":lines cs
lines (c:cs) =
    case lines cs of
        [] -> [[c]]
        (l:ls) -> (c:l):ls
```

3.4.3 Guards

Jedes pattern matching kann eine zusätzliche boolesche Bedingung bekommen, welche *Guard* genannt wird:

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

Damit kann man beispielsweise die ersten n Elemente einer Liste extrahieren:

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take (n+1) (x:xs) = x:take n xs
```

Guards sind auch bei `let`, `where` und `case` erlaubt.

3.5 Funktionen höherer Ordnung

Funktionen sind Bürger erster Klasse: Sie können wie alle anderen Werte verwendet werden. Anwendungen davon sind:

- generische Programmierung
- Programmschemata (Kontrollstrukturen)

Wir erreichen damit eine bessere Wiederverwendbarkeit und eine höhere Modularität des Codes.

3.5.1 Beispiel: Ableitungsfunktion

Die Ableitungsfunktion ist eine Funktion, die zu einer Funktion eine neue Funktion liefert. Die numerische Berechnung sieht so aus:

$f'(x) = \lim_{dx \to 0} \frac{f(x+dx)-f(x)}{dx}$

Eine Implementierung mit kleinem dx könnte in Haskell so aussehen:

```
dx = 0.0001

ableitung :: (Float -> Float) -> (Float -> Float)
ableitung f = f'
 where f' :: Float -> Float
       f' x = (f (x + dx) - f x) / dx
```

Nun wird `(ableitung sin) 0.0` ausgewertet zu `1.0`, `(ableitung square) 1.0` wird ausgewertet zu 2.00033.

3.5.2 Anonyme Funktionen (Lambda-Abstraktionen)

`ableitung (\x -> x * x)` entspricht der Funktion $x \mapsto 2x$. Hier steht \ für das λ, `x` ist ein Parameter und `x * x` ein Ausdruck.

Allgemein formuliert man anonyme Funktionen in Haskell wie folgt:

$\backslash p_1 ... p_n$ `-> e`

wobei $p_1, ..., p_n$ Pattern sind, und `e` ein Ausdruck.

Dann können wir auch schreiben:

```
ableitung f = \x -> (f (x + dx) - f x) / dx
```

Aber `ableitung` ist nicht die einzige Funktion mit funktionalem Ergebnis: So kann man zum Beispiel die Funktion `add` auf drei verschiedene Weisen definieren:

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

oder

```
add = \x y -> x + y
```

oder

```
add x = \y -> x + y
```

Also kann `add` auch als Konstante gesehen werden, die eine Funktion als Ergebnis liefert, oder als Funktion, die einen `Int` nimmt und eine Funktion liefert, die einen weiteren `Int` nimmt und erst dann einen `Int` liefert.

Somit müssen die Typen `Int -> Int -> Int` und `Int -> (Int -> Int)` identisch sein. Die Klammerung ergibt sich durch rechtsassoziative Bindung des Typkonstruktors `(->)`. Beachte aber, dass `(a -> b) -> c` *nicht* das gleiche ist wie `a -> b -> c` oder `a -> (b -> c)`!

Es wäre also unabhängig von der Definition von `add` sinnvoll, folgendes zu schreiben:

```
ableitung (add 2)
```

Wird eine Funktion auf „zu wenige“ Argumente appliziert, nennt man dies *partielle Applikation*. Die partielle Applikation wird syntaktisch einfach möglich durch Currying. *Currying* geht auf *Haskell B. Curry* und *Schönfinkel* zurück, die in den 40er Jahren unabhängig voneinander folgende Isomorphie festgestellt haben:

$$[A \times B \to C] \simeq [A \to (B \to C)]$$

Mit Hilfe von partieller Applikation lassen sich nun eine Rehie von Funktionen definieren:

- `take 42 :: [a] -> [a]` liefert die bis zu 42 ersten Elemente einer Liste.
- `(+) 1 :: Int -> Int` ist die Inkrementfunktion.

Bei Operatoren bieten die sog. *Sections* eine zusätzliche, verkürzte Schreibweise:

- `(1+)` steht für die Inkrementfunktion.
- `(2-)` steht für `\x -> 2-x`.
- `(/2)` steht für `\x -> x/2`.
- `(-2)` steht aber *nicht* für `\x -> x-2`, da der Compiler hier das Minus-Zeichen nicht vom unären Minus unterscheiden kann.

Es ist auch partielle Applikation auf das zweite Argument möglich:

```
(/b) a = (a/) b = a / b
```

Die Reihenfolge der Argumente ist wegen partieller Applikation also eine Designentscheidung, aber mit λ-Abstraktion und der Funktion `flip` bei der partiellen Anwendung in anderer Reihenfolge noch veränderbar.

3.5.3 Generische Programmierung

Wir betrachten die folgenden Funktionen `incList` und `codeStr`:

```
incList :: [Int] -> [Int]
incList [] = []
incList (x:xs) = (x + 1):incList xs

code :: Char -> Char
code c | c == 'Z' = 'A'
       | c == 'z' = 'a'
       | otherwise = chr (ord c + 1)

codeStr :: String -> String
codeStr "" = ""
codeStr (c:cs) = code c : codeStr cs
```

Dann wird `codeStr ‘‘Informatik"` auswewertet zu `"Jogpsnbujl"`. Wir können beobachten, dass es sich bei `incList` und `codeStr` um fast identische Definitionen handelt, die sich nur in der Funktion unterscheiden, die auf die Listenelemente angewandt wird.

Die Verallgemeinerung ist die Funktion `map`:

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

Damit lassen sich `incList` und `codeStr` viel einfacher ausdrücken:

```
incList = map (+1)
codeStr = map code
```

Wir betrachten zwei weitere Beispiele: Eine Funktion, die die Summe aller Zahlen in einer Liste liefert, und eine Funktion, die zur Eingabekontrolle die Summe der ASCII-Werte einer Zeichenkette berechnet:

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

checkSum :: String -> Int
checkSum "" = 1
checkSum (c:cs) = ord c + checkSum cs
```

Findet sich hier ebenfalls ein gemeinsames Muster? Ja! Beide Funktionen lassen sich viel einfacher mit der mächtigen Funktion `foldr` ausdrücken:

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

sum = foldr (+) 0
checkSum = foldr (\\c res -> ord c + res) 1
```

Zum Verständnis von `foldr` hilft die folgende Sichtweise: Die an `foldr` übergebene Funktion vom Typ `(a -> b -> b)` wird als Ersatz für den Listenkonstruktor `(:)` in der Liste eingesetzt, und das übergebene Element vom Typ `b` als Ersatz für die leere Liste `[]`. Man beachte: Der Typ der übergebenen Funktionen passt zum Typ von `(:)`.

So entsprichen sich also die folgenden Ausdrücke:

```
foldr f e [1,2,3]
foldr f e ((:) 1 ((:) 2 ((:) 3 [])))
          (f   1 (f   2 (f   3 e)))
```

Das allgemeine Vorgehen beim Entwerfen solcher Funktionen ist das Folgende: Suche ein allgemeines Schema und realisiere es durch funktionale Parameter.

Ein weiteres Schema kennen wir von der Funktion `filter`:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

Diese können wir zum Beispiel verwenden, um eine Liste in eine Menge umzuwandeln, also um alle doppelten Einträge zu entfernen:

```
nub :: [Int] -> [Int]
nub [] = []
nub (x:xs) = x:(nub (filter (/= x) xs))
```

Mit Hilfe von `filter` können wir sogar Listen mittels Quicksort sortieren:

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) =
    qsort (filter (<= x) xs) ++ [x] ++ qsort (filter (> x) xs)
```

Auch `filter` kann mit Hilfe von foldr definiert werden:

```
filter p = foldr (\x ys -> if p x then x:ys else ys) []
```

In der Tat ist `foldr` ein sehr allgemeines Skelett, es entspricht dem Katamorphismus der Kategorientheorie.

`foldr` hat manchmal aber auch Nachteile: So führt

```
foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))
```

zu einer sehr großen Berechnung, die zunächst auf dem Stack aufgebaut und erst zum Schluss ausgerechnet wird.

Eine bessere Lösung finden wir mit Hilfe der Akkumulatortechnik:

```
sum xs = sum' xs 0
  where sum' :: [Int] -> Int -> Int
        sum' [] s = s
        sum' (x:xs) s = sum' xs (x + s)
```

Damit wird ein Aufruf von `sum [1,2,3]` ersetzt durch `((0 + 1) + 2) + 3`, was direkt ausgerechnet werden kann.

Aber auch das ist als fold-Variante möglich:

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

Damit wird ein Aufruf von `foldl f e (`x_1 `:` x_2 `: ... :` x_n `: [])` ersetzt durch `f ... (f (f e` x_1`)` x_2`)` ... x_n

Jetzt können wir `sum` natürlich wieder viel einfacher definieren:

```
sum = foldl (+) 0
```

3.5.4 Kontrollstrukturen

Viele der Kontrollstrukturen, die wir aus anderen Programmiersprachen kennen, lassen sich auch in Haskell modellieren. Wir betrachten zum Beispiel die `while`-Schleife:

```
x = 1;
while x < 100 do x = 2*x od
```

Eine `while`-Schleife besteht im Allgemeinen aus:

- dem Zustand vor der Schleife (Anfangswert)
- einer Bedingung
- einem Rumpf für die Zustandsänderung

In Haskell sieht das wie folgt aus:

```
while :: (a -> Bool) -> (a -> a) -> a -> a
while p f x | p x = while p f (f x)
            | otherwise = x
```

Dann wird `while (<100) (2*) 1` ausgewertet zu `128`.

Man beachte, dass es sich hierbei um keine Spracherweiterung handelt! Diese Kontrollstruktur ist nichts anderes als eine Funktion, ein Bürger erster Klasse.

3.5.5 Funktionen als Datenstrukturen

Was sind Datenstrukturen? Abstrakt betrachtet sind Datenstrukturen Objekte mit bestimmten Operationen:

- Konstruktoren (wie `(:)` oder `[]`)
- Selektoren (wie `head` oder `tail`, und pattern matching)
- Testfunktionen (wie `null`, und pattern matching)
- Verknüpfungen (wie `++`)

Wichtig ist die dabei die Funktionalität, also die Schnittstelle, nicht die Implementierung. Eine Datenstruktur entspricht somit einem Satz von Funktionen.

Als Beispiel möchten wir Arrays (Felder mit beliebigen Elementen) in Haskell implementieren.

Die Konstruktoren haben folgenden Typ:

```
emptyArray :: Feld a
putIndex :: Feld a -> Int -> a -> Feld a
```

Wir benötigen nur einen einzigen Selektor:

```
getIndex :: Feld a -> Int -> a
```

Eine Implementierung als Funktion sieht dann zum Beispiel so aus:

```
type Feld a = Int -> a

emptyArray i =
    error("Zugriff auf nicht initialisierte Arraykomponente " ++ show i)

getIndex a i = a i

putIndex a i v = a'
  where a' j | i == j = v
             | otherwise = a i
```

Der Vorteil dieser Implementierung ist ihre konzeptionelle Klarheit: Sie entspricht genau der Spezifikation. Ihr Nachteil liegt darin, dass die Zugriffszeit abhängig von der Anzahl der vorangegangenen `putIndex`-Aufrufe ist.

3.5.6 Wichtige Funktionen höherer Ordnung

Eine wichtige Funktion höherer Ordnung ist die Verkettung von Funktionen `(.)`:

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

Eine weitere ist die bereits erwähnte Funktion `flip`, mit deren Hilfe sich die Reihenfolge der Parameter einer Funktion vertauschen lässt:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f = \x y -> f y x
```

Diese kann man zum Beispiel auf die Funktion `map` anwenden, um die bearbeitende Liste zuerst angeben zu können:

```
(flip map) :: [a] -> (a -> b) -> [b]
(flip map) [1,2] :: (Int -> b) -> [b]
```

Zwei weitere interessante Funktionen höherer Ordnung sind die Funktionen `curry` und `uncurry`. Diese erlauben die Anwendung von Funktionen, die auf Tupeln definiert sind, auf einzelne Elemente, und umgekehrt:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Zuletzt betrachten wir noch die Funktion `const`, die zwei Argumente nimmt und das erste zurückgibt:

```
const :: a -> b -> a
const x _ = x
```

3.6 Typklassen und Überladung

Wir betrachten die Funktion `elem`, die überprüft, ob ein Element in einer Liste enthalten ist:

```
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

Wie sind nun mögliche Typen von `elem`? Zum Beispiel:

```
Int -> [Int] -> Bool
Bool -> [Bool] -> Bool
Char -> String -> Bool
```

Aber leider nicht `a -> [a] -> Bool`, da `a` zu allgemein ist: `a` beinhaltet z.B. auch Funktionen, auf denen Gleichheit aber nicht definiert werden kann. Wir benötigen also eine Einschränkung auf Typen, für die Gleichheit definiert ist. Diese erreichen wir in Haskell so:

```
elem :: Eq a => a -> [a] -> Bool
```

`Eq a` nennt man einen *Typconstraint*, der a einschränkt. Möchte man mehrere Typconstraints angeben, muss man ggf. klammern.

Die Klasse `Eq` ist in Haskell wie folgt definiert:

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

In einer *Klasse* werden mehrere Typen zusammengefasst, für die die Funktionen der Klasse definiert sind. Bei `Eq` sind das zum Beispiel die Funktionen `(==)` und `(/=)`.

Typen werden zu *Instanzen* einer Klasse durch:

```
data Tree = Empty | Node Tree Int Tree

instance Eq Tree where
    Empty == Empty = True
    (Node tl n tr) == (Node tl' n' tr') =
        tl == tl' && n == n' && tr == tr'
    _ == _ = False

    t1 /= t2 = not (t1 == t2)
```

Dann ist `(==)` und `(/=)` für den Typ `Tree` verwendbar.

Und was ist mit polymorphen Typen?

```
data Tree a = Empty | Node (Tree a) Int (Tree a)

instance Eq a => Eq (Tree a) where
    ...
```

Beachte: So werden unendlich viele Typen Instanz der Klasse `Eq`.

3.6.1 Vordefinierte Funktionen in einer Klasse

Die Definition von `(/=)` wird in fast jeder Instanzdefinition so wie oben aussehen.

Deshalb ist häufig eine Vordefinition in der Klassendefinition sinnvoll, welche aber in der Instanzdefinition überschrieben werden kann oder sogar muss:

```
class Eq a where
    (==),(/=) :: a -> a -> Bool
    x1 == x2 = not (x1 /= x2)
    x1 /= x2 = not (x1 == x2)
```

3.6.2 Erweiterung von Klassen

Für manche Typen macht es auch Sinn, eine totale Ordnung zu definieren. Diese finden wir in Haskell in einer Erweiterung von `Eq`:

```
data Ordering = LT | EQ | GT

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

    ... -- vordefinierte Implementierungen
```

Eine minimale Instanzdefinition benötigt zumindest `compare` oder `(<=)`. Weitere vordefinierte Klassen sind `Num`, `Show` und `Read`:

- `Num`: stellt Zahlen zum Rechnen dar (`(+) :: Num a => a -> a -> a`)
- `Show`: zum Verwandeln von Werten in Strings (`show :: Show a => a -> String`)
- `Read`: zum Konstruieren von Werten aus Strings (`read :: Read a => String -> a`)

Noch mehr vordefinierte Klassen werden im Master-Modul „Funktionale Programmierung“ vorgestellt.

Automatische Instantiierung vordefinierter Klassen (außer `Num`) erreicht man mittels `deriving` $(\kappa_1, ..., \kappa_n)$ hinter der Datentypdefinition.

Aufgabe: Überprüfen Sie die Typen aller in der Vorlesung definierten Funktionen auf Ihren allgemeinsten Typ! Diese lauten zum Beispiel:

```
(+) :: Num a => [a] -> [a]
nub :: Eq a => [a] -> [a]
qsort :: Ord a  => [a] -> [a]
```

3.6.3 Die Klasse `Read`

Wir betrachten folgende Funktionsdefinition zum Parsen von Strings in Werte:

```
type ReadS a = String -> [(a,String)]
```

Was hat man sich hier beim Rückgabetyp von `ReadS` gedacht? Der erste Teil des Tupels ist das Parsergebnis, der zweite Teil ist der noch zu parsende Reststring: Betrachtet man zum Beispiel den String `Node Empty 42 Empty"`, so wird schnell klar, dass nach dem Lesen der Anfangszeichenkette `Node` ein Baum folgen muss. Dann möchten wir uns den verbleibenden, noch zu parsenden Reststring erhalten, um ihn später zu betrachten.

Falls außerdem mehrere Ergebnisse möglich sind, werden diese als Liste zurückgegeben. Lässt sich der übergebene String nicht parsen, so ist der zurückgegebene Wert die leere Liste.

Das kann in der Anwendung so aussehen:

```
class Read a where
    readsPrec :: Int -> ReadS a
    readList :: ReadS [a] -- vordefiniert
```

Dann sind zwei Funktionen `reads` und `read` definiert wie folgt:

```
reads :: Read a => ReadS a
reads = readsPrec 0

read :: Read a => String -> a
read str = case reads str of
    [(x,"")] -> x
    _ -> error "no parse"
```

Einige Auswertungen von Aufrufen von `reads` und `read` sehen dann zum Beispiel so aus:

```
reads "(3,'a')" :: [((Int,Char),String)]
    = [((3,'a'),"")]

reads "(3,'a')" :: [((Int,Int),String)]
    = []

read "(3,'a')" :: (Int,Char)
    = (3,'a')

read "(3,'a')" :: (Int,Int)
    = error: no parse

reads "3,'a')" :: [(Int,String)]
    = [(3,",'a'")]
```

3.7 Lazy Evaluation

Wir betrachten das Haskellprogramm

```
f x = 1
h = h
```

und die Anfrage `f h`: Dann kann stets zuerst das `f` ausgewertet werden, was zum Ergebnis `1` führt, oder es wird zunächst versucht, `h` auszuwerten - was nie termiert.

Nicht jeder Berechnungspfad terminiert also. Wir unterscheiden zwei ausgezeichnete Reduktionen:

- *left-most-inner-most (LI)*: applikative Ordnung (strikte Funktionen)
- *left-most-outer-most (LO)*: Normalordnung (nicht-strikte Funktionen)

Ein Vorteil der LO-Reduktion liegt darin, dass sie berechnungsvollständig ist: Alles, was irgendwie berechnet werden kann, wird auch berechnet. Allerdings kann LO auch ineffizient sein, da Berechnungen verdoppelt werden können.

Kann LO trotzdem auch in der Praxis Vorteile bringen? Ja! Sie bietet

- Vermeidung überflüssiger (ggf. unendlicher) Berechnungen
- Rechnen mit unendlichen Datenstrukturen

Zum Beispiel definiert folgende Funktion `from` die unendliche Liste natürlicher Zahlen, die mit `n` beginnt:

```
from :: Num a => a -> [a]
from n = n : from (n + 1)

take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x:take (n-1) xs
```

`take 3 (from 1)` wird ausgewertet zu `[1,2,3]`, denn LO liefert:

```
  take 1 (from 1)
= take 1 (1:from 2)
= 1 : take 0 (from 2)
= 1 : []
```

Der Vorteil liegt in der Trennung von Kontrolle (`take 3`) und Daten (`from 1`).

Betrachten wir zum Beispiel die Primzahlberechnung mittels Sieb des Eratosthenes. Die Idee lautet wie folgt:

1. Nehme die Liste aller Zahlen größer oder gleich 2.
2. Streiche alle Vielfachen der ersten (Prim-)Zahl.
3. Das erste Listenelement ist Primzahl. Fahre bei 2.) mit Restliste fort.

Dies lässt sich in Haskell zum Beispiel so implementieren:

```
sieve :: [Int] -> [Int]
sieve (p:xs) = p:sieve (filter (\x -> x 'mod' p > 0) xs)

primes :: [Int]
primes = sieve (from 2)
```

Das Argument von `sieve` ist eine Eingabeliste, die mit einer Primzahl beginnt und in der alle Vielfachen kleinerer Primzahlen fehlen. Das Ergebnis ist eine Liste aller Primzahlen!

Jetzt liefert ein Aufruf von `take 10 primes` die ersten zehn Primzahlen: `[2,3,5,7,11,13,17,19,23,27]`. Und mit Hilfe von `(!!)` können wir uns die zehnte Primzahl direkt ausgeben lassen: `primes!!10` wird ausgewertet zu `27`.

Die Programmierung mit unendlichen Datenstrukturen kann auch als Alternative zur Akkumulatortechnik verwendet werden:

Wir nehmen als Beispiel die Fibonaccifunktion. Um die n-te Fibonaccizahl zu erhalten, erzeugen wir die Liste aller Fibonaccizahlen und schlagen das n-te Element nach:

```
fibgen :: Int -> Int -> [Int]
fibgen n1 n2 = n1:fibgen n2 (n1 + n2)

fibs :: [Int]
fibs = fibgen 0 1

fib :: Int -> Int
fib n = fibs!!n
```

Dann wird `fib 10` ausgewertet zu `34`.

Ein Nachteil der LO-Strategie bleibt jedoch: Berechnungen können dupliziert werden. Wir betrachten erneut die einfache Funktion `double`:

```
double x = x + x
```

Übergeben wir dieser Funktion jetzt `double 3` als Argument, dann sieht die Auswertung nach der LI-Strategie so aus:

```
  double (double 3)
= double (3 + 3)
= double    6
= 6 + 6
= 12
```

Nach der LO-Strategie ergibt sich statt dessen:

```
  double (double 3)
= double 3 + double 3
= (3 + 3)  + double 3
=    6     + double 3
=    6     + (3 + 3)
=    6     +    6
=          12
```

Wegen der offensichtlichen Ineffizienz verwendet keine Programmiersprache verwendet die LO-Strategie.

Eine Optimierung der Strategie führt zur *Lazy-Auswertung*, bei der statt Termen Graphen reduziert werden. Variablen des Programms entsprechen dann Zeigern auf Audrücke, und die Auswertung eines Ausdrucks gilt für alle Variablen, die auf diesen Ausdruck verweisen: Dies bezeichnet man als *sharing*. Normalisierung führt dazu Variablen für jeden Teilausdruck ein. Für obiges Beispiel sieht das aus wie folgt:

```
double x = x + x

main = let y = 3
           z = double y
       in double z
```

Dann verläuft die Auswertung so, wie auf Abbildung 3.3 dargestellt. Die schwarzen Linien zeigen dabei jeweils einen Reduktionsschritt an, blaue Linien sind Zeiger auf Ausdrücke.

Formalisiert wurde diese Strategie im Jahre 1993 von Launchburg. Die Lazy-Auswertung ist optimal bzgl. Länge der Auswertung: Es erfolgt keine

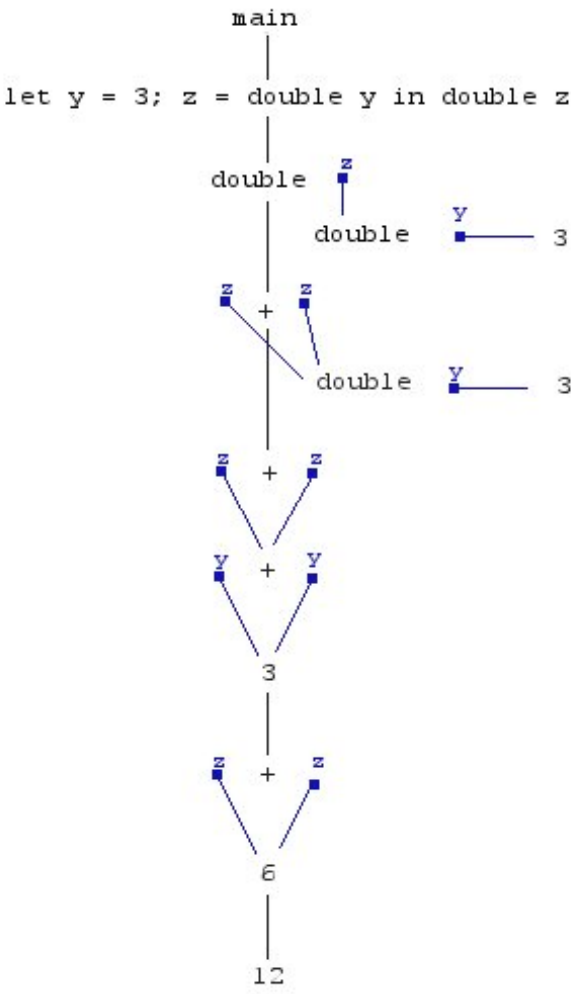

Abbildung 3.3: Sharing bei lazy evaluation

überflüssige Berechnung wie bei der LI-Strategie, und keine Duplikation wie bei der LO-Strategie. Allerdings benötigt sie manchmal viel Speicher.

In der Programmiersprache Haskell finden wir diese Lazy-Auswertung, in den Sprachen ML, Erlang, Scheme und Lisp finden wir hingegen die LI-Strategie.

Ein weiterer Vorteil der Lazy-Auswertung ist die schöne Komponierbarkeit von Funktionen: Angenommen, wir hätten eine Generator-Funktion, z.B. vom Typ `-> String`, und eine Konsumenten-Funktion, z.B. vom Typ `String -> Datei`.

Durch laziness entstehen hierbei keine großen Zwischendatenstrukturen, sondern nur die Teile, welche zur gegebenen Zeit benötigt werden, und auch diese werden danach direkt wieder freigegeben.

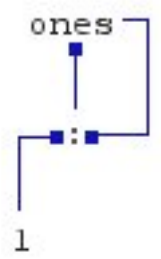

Abbildung 3.4: Zyklische Liste `ones`

Haskell erlaubt auch *zyklische Datenstrukturen* wie z.B. die Liste `ones = 1:ones`. Diese kann sogar mit konstantem Speicherbedarf dargestellt werden.

3.8 Ein- und Ausgabe

Haskell ist eine reine funktionale Sprache, d.h. Funktionen haben keine Seiteneffekte. Wie kann Ein- und Ausgabe in solch eine Sprache integriert werden?

Die erste Idee lautet: Wie in ML, Erlang oder Scheme trotzdem als Seiteneffekte.

```
main = let str = getLine
       in putStr str
```

Hier soll `getLine` eine Zeile von der Tastatur einlesen und `putStr` einen String auf der Standardausgabe ausgeben. Was könnte der Typ von `main` oder `putStr` sein?

In Haskell haben wir als kleinsten Typ `()` (sprich: unit), dessen einziger Wert `()::()` ist (entspricht `void` in Sprachen wie Java). Hat `main` aber als Ergebnistyp `()`, dann ist wegen laziness doch eigentlich keine Ein- oder Ausgabe notwendig, um das Ergebnis `()` berechnen zu können...

Deshalb muss `putStr` als Seiteneffekt den übergebenen String ausgeben, bevor das Ergebnis `()` zurückgegeben wird.

Wir betrachten ein weiteres Beispiel:

```
main = let x = putStr "Hi" in
       x; x
```

Hier soll `;` bewirken, dass `Hi` zweimal hintereinander ausgegeben wird. Das Problem dabei ist jedoch, dass wegen der laziness `x` nur einmal berechnet

wird, was wiederum bedeutet, dass der Seiteneffekt, die Ausgabe, nur einmal durchgeführt wird.

Ein weiteres Problem ergibt sich bei folgendem häufigen Szenario:

```
main = let dataBase = readDBFromUser
           request = readRequestFromUser
       in print (lookup request dataBase)
```

Hier verhindert die laziness die gewünschte Eingabereihenfolge. All diese Probleme werden in Haskell mit Monaden gelöst.

3.8.1 I/O-Monade

Die Ein- und Ausgabe erfolgt in Haskell *nicht* als Seiteneffekt. I/O-Funktionen liefern eine Aktion der Ein- oder Ausgabe als Ergebnis, also als Wert. `putStr "Hi"` ist eine Aktion, welche `Hi` auf die Standardausgabe schreibt.

Eine Aktion ist im Prinzip eine Abbildung `Welt -> Welt`. Diesen Typ nennen wir abstrakt `IO ()`.

```
putStr :: String -> IO ()
putChar :: Char -> IO ()
```

IO-Aktionen sind first class citizens, sie können damit z.B. in Datenstrukturen gespeichert werden. Ein Programm definiert eine große IO-Aktion, welche auf die initiale Welt angewandt wird und abschließend eine veränderte Welt liefert (`main :: IO ()`).

Das Zusammensetzen von IO-Aktionen erreicht man mit dem Sequenzoperator `(>>) :: IO () -> IO () -> IO ()`.

Dann sind äquivalent:

```
main = putStr "Hi" >> putStr "Hi"
```

```
main = let pHi = putStr "Hi"
       in pHi >> pHi
```

```
main = let l = repeat (putStr "Hi")
       in l!!0 >> l!!42
```

Hier liefert `repeat :: a -> [a]` eine unendliche Liste gefüllt mit der übergebenen Aktion.

Wie kombiniert man dies aber mit rein funktionalen Berechnungen?

```
fac :: Int -> Int
-- ...

main = putStr (show (fac 42))
main = print (fac 42)
```

Dabei ist `print` eine Funktion, die einen übergebenen Wert zuerst in eine Zeichenkette umwandelt und dann auf die Standardausgabe schreibt: `print :: Show a => a -> IO ()`.

Wir betrachten noch ein Beispiel: Folgende Definitionen von `putStr` sind äquivalent.

```
putStr :: String -> IO ()
putStr "" = return ()
putStr (c:cs) = putChar c >> putStr cs

putStr = foldr (\c -> (putChar c >>)) (return ())
```

`return ()` ist hier sozusagen die leere IO-Aktion.

Und wie ist jetzt Eingabe in Haskell möglich? `IO` ist tatsächlich ein Typkonstruktor, dessen Argument den Typ des Rückgabewertes der Aktion darstellt:

```
getChar :: IO Char
getLine :: IO String
return :: a -> IO a
```

Wie kann das Ergebnis einer IO-Aktion an eine nachfolgende IO-Aktion übergeben werden? Hierzu verwendet man den Bind-Operator:

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

In der Verwendung sieht das zum Beispiel so aus:

```
getChar >>= putChar
```

Hier hat `getChar` den Typ `IO Char`, `putChar` hat den Typ `Char -> IO ()`: Also hat `getChar >>= putChar` den Typ `IO ()`. Es liest ein Zeichen ein und gibt es wieder aus.

Als nächstes möchten wir eine ganze Zeile einlesen:

```
getLine :: IO String
getLine =
    getChar >>= \c ->
        if c == '\n'
            then return ""
            else getLine >>= \cs -> return (c:cs)
```

Wenn man genauer hinsieht, dann ähnelt die Zeichenfolge `>>=`
eine Zuweisung in einer imperativen Sprache, nur dass die linke und die rechte Seite vertauscht ist.

3.8.2 `do`-Notation

Mit `do {` $a_1; ...; a_n$ `}` oder

```
do
a1
...
an
```

steht dem/der Programmierer/in eine alternative, „imperative“ Notation zur Verfügung. Hierbei ersetzt `p <-` e_1`;` e_2 den Bind-Aufruf e_1 `>>=`
`p ->` e_2:

```
getLine = do
    c <- getChar
    if c == '\n'
        then return ""
        else do
            cs <- getLine
            return (c:cs)
```

3.8.3 Ausgabe von Zwischenergebnissen

Wir möchten eine Funktion schreiben, die die Fakultät berechnet, und dabei alle Zwischenergebnisse der Berechnung ausgibt.

```
fac :: Int -> IO Int
fac 0 = return 1
fac (n+1) = do
    rn <- fac n
    print rn
```

```
    return ((n+1) * rn)

main :: IO ()
main = do
    putStrLn "n: "
    str <- getLine
    facn <- fac (read str)
    putStrLn ("Factorial: " ++ show facn)
```

Die Verwendung sieht dann so aus:

```
> main
n: 6
1
2
6
24
120
Factorial: 720
```

Aber solche Programme sollte man vermeiden! Es ist immer besser, Ein- und Ausgabe auf der einen Seite und Berechnungen auf der anderen Seite zu trennen. Als gängiges Schema hat sich etabliert:

```
main = do
    input <- getInput
    let res = computation input
    print res
```

Hier ist die Zeile `let res = computation input` eine rein funktionale Berechnung. Das `let` benötigt im `do`-Block kein `in`.

3.9 List Comprehensions

Haskell stellt noch ein wenig syntaktischen Zucker für Listendefinitionen zur Verfügung: So lässt sich die Liste der ganzen Zahlen von eins bis vier in Haskell mit `[1..4]` beschreiben. Aber man kann auch komplexere Muster vorgeben: `[3,1..]` wird zum Beispiel zu der unendlichen Liste ausgewertet, die mit drei beginnt und dann immer mit dem Vorvorgänger fortfährt: `[3,1,-1,-3,-5,...]`.

Wir können sogar Listen mit einer an die Mathematik angelehnten Notation beschreiben: So liefert der Ausdruck

```
[(i,j) | i <- [1..3], j <- [2,3,4], i /= j]
```

die Liste `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]`. Erlaubt sind dabei Generatoren wie `i <- [1..3]` und boolesche Bedingungen wie `i /= j`. Auch `let` ist erlaubt.

`[[0..n] | n <- [0..]]` liefert also alle endlichen Anfangsfolgen der Menge der natürlichen Zahlen.

Abschließend betrachten wir noch eine andere Definition der Funktion `concat`, die aus einer Liste von Listen eine Ergebnisliste mit deren Einträgen macht:

```
concat :: [[a]] -> [a]
concat xss = [y | ys <- xss, y <- ys]
```

Kapitel 4

Einführung in die Logikprogrammierung

4.1 Motivation

Für die Logikprogrammierung sprechen ähnliche Gründe wie für die Funktionale Programmierung:

- Abstraktion vom Rechner
- Programme: mathematische Objekte, Relationen statt Funktionen
- Computer soll *Lösungen* finden, nicht nur Werte berechnen
- weniger Aussagen über Berechnungsrichtung
- flexible Benutzung von Relationen

Als Einstiegsbeispiel möchten wir einmal Verwandschaftsbeziehungen implementieren. Ziel ist die Berechnung von Antworten auf Fragen wie „Welche Großväter hat Andreas?“ und ähnliche andere Fragen.

Unser konkretes Beispiel sieht so aus: Christine ist verheiratet mit Heinz. Christine hat zwei Kinder: Herbert und Angelika. Herbert ist verheiratet mit Monika. Monika hat zwei Kinder: Susanne und Norbert. Maria ist verheiratet mit Fritz. Maria hat ein Kind: Hubert. Angelika ist verheiratet mit Hubert. Angelika hat ein Kind: Andreas.

In der funktionalen Programmiersprache Haskell lässt sich dieser Sachverhalt zum Beispiel wie folgt modellieren:

```
data Person = Christine | Heinz | ... | Andreas

ehemannn :: Person -> Person
ehemann Christine = Heinz
```

```
ehemann Maria = Fritz
ehemann Monika = Herbert
ehemann Angelika = Hubert

mutter :: Person -> Person
mutter Herbert = Christine
...
mutter Andreas = Angelika

vater :: Person -> Person
vater kind = ehemann (mutter kind)

grossvater :: Person -> Person -> Bool
grossvater e g | g == vater (vater e) = True
               | g == vater (mutter e) = True
```

Jetzt finden sich schnell Antworten auf Fragen wie „Wer ist Vater von Norbert?“ oder „Ist Heinz Großvater von Andreas?“:

```
> vater Norbert
Herbert

> grossvater Andreas Heinz
True
```

Folgende Fragen kann unser Programm aber nicht direkt beantworten:

- Welche Kinder hat Herbert?
- Welche Enkel hat Heinz?

Dies wäre möglich, falls Variablen in Ausdrücken zulässig wären: Dann würde z.B. `vater k == Herbert` liefern: `k = Susanne oder k = Norbert`. Oder `grossvater e Heinz` würde liefern: `True falls e = Susanne oder e = Norbert oder e = Andreas`.

Genau dies ist in Logiksprachen wie Prolog erlaubt. Diese haben folgende Charakterisitik:

- „Freie“ („logische“) Variablen in Ausdrücken (auch Regeln) erlaubt
- Berechnung von *Lösungen* (Werte für freie Variablen)
- Problem: Wie konstruktiv berechnen? → später

- Berechnungsprinzip: Schlussfolgerungen aus gegebenem Wissen ziehen

Ein *Prolog-Programm* ist eine Menge von Fakten und Regeln für Prädikate, also Aussagen über die Objekte. Eine *Aussage* wiederum besteht aus

1. der Art der Aussage (Eigenschaft): 3 ist *Primzahl*
2. den beteiligten Objeken: *3* ist Primzahl.

Diese gibt man in Prolog in der Standardpräfixschreibweise an:
`name(`$objekt_1, ..., objekt_n$`)`, z.B. `primzahl(3)`. Dabei werden alle Atome klein geschrieben, und die Reihenfolge der Objekte ist durchaus relevant.

Als *Fakten* bezeichnen wir Aussagen, die als richtig angenommen werden. Syntaktisch werden diese durch einen Punkt und ein whitespace am Ende abgeschlossen, vor der öffnenden Klammer darf kein Leerzeichen stehen:

```
ehemann(christine,heinz).
ehemann(maria,fritz).
...

mutter(herbert,christine).
mutter(angelika,christine).
...
```

Fakten alleine entsprechen einem Notizbuch oder einer relationalen Datenbank.

Regeln sind jetzt Schlussfolgerungen zur Ableitung neuer Aussagen: *Wenn* Aussage1 und Aussage2 richtig sind, *dann* ist Aussage3 richtig. Dies schreiben wir in Prolog so: `Aussage3 :- Aussage1, Aussage2.` Hier steht das Komma `,` für das logische Und ($\wedge$), `:-` steht für einen Schlussfolgerungspfeil ($\Leftarrow$).

So lautet zum Beispiel die Regel „Der Vater von Susanne ist Herbert, falls Herbert Ehemann von Monika und Monika Mutter von Susanne ist.“ in Prolog:

```
vater(susanne,herbert) :-
    ehemann(monika,herbert),
    mutter(susanne,monika).
```

Diese Regel ist natürlich sehr speziell, aber Prolog erlaubt hier zum Glück eine Verallgemeinerung. Wir geben unbekannte Objekte statt fester Namen an: *Variablen*. Diese beginnen mit Großbuchstaben und stehen für beliebige Objekte. Regeln mit Variablen entsprechen unendlich vielen konkreten Regeln.

So ergeben sich zum Beispiel folgende Regeln für Vater- und Großvaterbeziehungen:

```
vater(Kind,Vater) :- ehemann(Mutter,Vater),mutter(Kind,Mutter).
grossvater(E,G) :- vater(E,V),vater(V,G).
grossvater(E,G) :- mutter(E,M),vater(M,G).
```

Fakten und Regeln zusammen entsprechen dem Wissen über ein Problem.

Nun können wir *Anfragen* an das Prolog-System stellen: Sind Aussagen wahr?

```
?- vater(norbert,herbert).
yes
?- vater(andreas,herbert).
no
```

In Anfragen sind auch Variablen erlaubt. Das Prolog-System versucht dann herauszufinden, für welche Werte an Stelle der Variablen die Aussage richtig ist.

Die Anfrage „Wer ist der Ehemann von Monika?“ sieht so aus:

```
?- ehemann(monika,Mann).
Mann = herbert
```

Und die Anfrage „Welche Enkel hat Heinz?“ so:

```
?- grossvater(Enkel,heinz).
Enkel = susanne
```

Die Eingabe eines Semikolons ; fordert das Prolog-System auf, weitere Lösungen zu suchen:

```
?- grossvater(Enkel,heinz).
Enkel = susanne;
Enkel = norbert;
Enkel = andreas;
no
```

Hier gibt das Prolog-System mit einem `no` zu verstehen, dass keine weiteren Lösungen gefunden wurden.

Ein Logikprogramm besteht also im Wesentlichen aus den folgenden Bestandteilen:

- *Fakten* (gültige Aussagen)
- *Regeln* (wenn-dann-Aussagen)
- *Anfragen* (Ist eine Aussage gültig?)
- *Variablen* (Für welche Werte ist eine Aussage gültig?)

Klauseln für ein *Prädikat* ergeben sich aus den Fakten und Regeln.

4.2 Syntax von Prolog

Es gibt in Prolog eine Reihe von Zeichenklassen:

- Großbuchstaben: A-Z
- Kleinbuchstaben: a-z
- Ziffern: 0-9
- Sonderzeichen:

```
+ - * / < = > ‘ \ : . ? @ # \$ & ^ ~ }
```

Zahlen, auch Gleitkommazahlen, sind nun Folgen von Ziffern.
Atome in Prolog bilden unzerlegbare Objekte:

- Folge von Kleinbuchstaben, Großbuchstaben, Ziffern, `-`, beginnend mit einem Kleinbuchstaben; oder
- Folge von Sonderzeichen; oder
- beliebige Sonderzeichen, eingefasst in `'`, z.B. `'ein Atom!'`; oder
- „Sonderatome“ (nicht beliebig verwendbar): `, ; ! []`

Konstanten sind Zahlen oder Atome.
Jetzt entstehen *Strukturen* durch Zusammenfassung mehrerer Objekte zu einem:

- *Funktor* (entspricht Konstruktor)
- *Komponenten*

Ein Beispiel für eine Struktur: `datum(1,6,27)`. Hier bezeichnet `datum` den Funktor, `1`, `6` und `27` sind die Komponenten. Zwischen Funktor und Komponenten darf kein Leerzeichen stehen.

Strukturen können auch geschachtelt werden: `person(fritz,meier,datum(1,6,27))`

Der Funktor ist dabei relevant: `datum(1,6,27)` ist nicht das Gleiche wie `zeit(1,6,27)`.

Listen sind in Prolog induktiv definiert durch:

- leere Liste: `[]`
- Struktur der Form `.(E,L)`, wobei `E` das erste Element der Liste darstellt und `L` die Restliste.

Eine Liste mit den Elementen `a`, `b` und `c` könnte also so aussehen:

```
.(a,.(b,.(c,[])))
```

Auch in Prolog gibt es Kurzschreibweisen für Listen: `[`E_1`,`E_2`,...,`E_n`]` steht für eine Liste mit den Elementen $E_1, E_2, ..., E_n$, `[E|L]` steht für `.(E,L)`.

Die folgenden Listen sind also äquivalent:

```
.(a,.(b,.(c,[])))
[a,b,c]
[a|[b,c]]
[a,b|[c]]
```

Texte werden in Prolog durch Listen von ASCII-Werten beschrieben: `Prolog` entspricht also `[80,114,111,108,111,103]`.

Auch in Prolog gibt es *Operatoren*: So lässt sich die „Summe von 1 und dem Produkt von 3 und 4“ beschreiben durch Strukturen: `+(1,*(3,4))`. Doch es gibt auch die natürliche Schreibweise: `1+3*4` beschreibt also das Gleiche.

Die *Operatorschreibweise* von Strukturen sieht in Prolog so aus:

1. Strukturen mit einer Komponente
   (a) *Präfixoperator*: `-2` entspricht `-(2)`
   (b) *Postfixoperator*: `2 fak fak` entspricht `fak(fak(2))`
2. Strukturen mit zwei Komponenten
   (a) *Infixoperator*: `2+3` entspricht `+(2,3)`

Bei Infixoperatoren entsteht natürlich sofort das Problem der Eindeutigkeit:

- `1-2-3` kann interpretiert werden als
  - `-(-(1,2),3)`: dann ist `-` linksassoziativ
  - `-(1,-(2,3))`: dann ist `-` rechtsassoziativ
- `12/6+1`
  - `+(/(12,6),1)`: `/` bindet stärker als `+`
  - `/(12,+(6,1))`: `+` bindet stärker als `/`

Der Prolog-Programmierer kann selbst Operatoren definieren. Dazu muss er Assoziativität und Bindungsstärke angeben. Dabei verwendet man die Struktur `op`; genaueres kann man unter dem Stichpunkt *Direktive* im Prolog-Handbuch nachlesen. Die üblichen mathematischen Operatoren wie z.B. `+ - *` sind bereits vordefiniert. Natürlich ist es auch immer möglich, Klammern zu setzen: `12/(6+1)`.

Variablen in Prolog werden durch eine Folge von Großbuchstaben, Ziffern und `_`, beginnend mit einem Großbuchstaben oder `_` beschrieben.

`datum(1,4,Jahr)` entspricht also allen ersten Apriltagen, `[A,B|L]` entspricht allen mindestens zwei-elementigen Listen.

Variablen können in einer Anfrage oder Klausel auch mehrfach Auftreten: So entspricht `[E,E|L]` allen Listen mit mindestens zwei Elementen, wobei die ersten beiden Elemente identisch sind.

Auch Prolog bietet *anonyme Variablen*: `_` repräsentiert ein Objekt, dessen Wert nicht interessiert. Hier steht jedes Vorkommen von `_` für einen anderen Wert. Als Beispiel greifen wir auf unser Verwandschaftsbeispiel zurück:

```
istEhemann(Person) :- ehemann(_, Person).
```

Die Anfrage `?- mutter(_,M).` fragt das Prolog-System nach allen Müttern.

Jede Konstante, Variable oder Struktur in Prolog ist ein *Term*. Ein *Grundterm* ist ein Term ohne Variablen.

Wir betrachten nun ein Beispiel zum Rechnen mit Listenstrukturen. Unser Ziel ist es, ein Prädikat `member(E,L)` zu definieren, das angibt „`E` kommt in der Liste `L` vor“. `member(E,L)` ist wahr, falls `E` das erste Element von `L` ist oder im Rest von `L` vorkommt. In Prolog sieht das aus wie folgt:

```
member(E,[E|_]).
member(E,[_|R]) :- member(E,R).
```

Nun können wir Anfragen an das Prolog-System stellen:

```
?- member(X,[1,2,3]).
X=1;
X=2;
X=3;
no
```

Es macht jetzt Sinn, die *Gleichheit von Termen* genauer unter die Lupe zu nehmen. Vergleichsoperatoren in Sprachen wie Java oder Haskell, z.B. `==`, beziehen sich immer auf die Gleichheit nach dem Ausrechnen der Ausdrücke auf beiden Seiten. In Prolog bezeichnet `=` hingegen die strukturelle Termgleichheit: Es wird nichts ausgerechnet!

```
?- 5 = 2+3.
no

?- datum(1,4,Jahr) = datum(Tag,4,2009).
Jahr = 2009
Tag = 1
```

Natürlich gibt es aber *Arithmetik in Prolog*: Ein aritmetischer Ausdruck ist eine Struktur mit Zahlen und Funktoren wie beispielsweise `+ - * /` oder `mod`. Vordefiniert ist zum Beispiel das Prädikat `is(X,Y)`, wobei `is` auch Infixoperator ist. `X is Y` ist gültig oder beweisbar, falls

1. `Y` zum Zeitpunkt des Beweises variablenfreier arithmetischer Ausdruck ist, und
2. `X=Z` gilt, wenn `Z` ausgerechneter Wert von Y ist

```
?- 16 is 5*3+1.
yes
?- X is 5*3+1.
X = 16
?- 2+1 is 2+1.
no
```

Eine Definition der Fakultätsfunktion in Prolog sieht also aus wie folgt:

`X =:= Y`	Wertgleichheit
`X =\= Y`	Wertungleichheit
`X < Y`	kleiner als
`X > Y`	größer als
`X >= Y`	größer gleich
`X =< Y`	kleiner gleich

Tabelle 4.1: Vergleiche in Prolog

```
fak(0,1).
fak(N,F) :- N > 0,
            N1 is N-1,
            fak(N1, F1), % Reihenfolge wichtig!
            F is F1 * N.
```

Bei arithmetischen Vergleichen in Prolog werden beide Argumente vor dem Vergleich mit `is` ausgewertet.

Das `is`-Prädikat ist partiell: Ist bei `X is Y` das `Y` kein variablenfreier arithmetischer Ausdruck, dann wird die Berechnung mit einer Fehlermeldung abgebrochen. Daher ist die Reihenfolge bei der Verwendung von `is` wichtig:

```
?- X=2, Y is 3+X. % links-rechts-Auswertung
Y = 5
X = 2
?- Y is 3 + X, X = 2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Die Arithmetik in Prolog ist also logisch unvollständig:

```
?- 5 is 3+2.
yes
?- X is 3+2.
X = 5
?- 5 is 3+X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Eine mögliche Lösung in Prolog ist das später genauer betrachtete CLP.

4.3 Elementare Programmiertechniken

4.3.1 Aufzählung des Suchraums

Beim Färben einer Landkarte mit beispielsweise vier Ländern hat man die vier Farben rot, gelb, grün und blau zur Verfügung und sucht eine Zuordnung, bei der aneinandergrenzende Länder verschiedene Farben haben. Die vier Länder seien wie folgt angeordnet:

- L1 grenzt an L2 und L3.
- L2 grenzt an L1, L3 und L4.
- L3 grenzt an L1, L2 und L4.
- L4 grenzt an L2 und L3.

Was *wissen* wir über das Problem?

1. Wir haben vier Farben:

```
farbe(rot).
farbe(gelb).
farbe(gruen).
farbe(blau).
```

2. Jedes Land hat eine dieser Farben:

```
faerbung(L1,L2,L3,L4) :- farbe(L1),farbe(L2),farbe(L3),farbe(L4).
```

3. Wann sind zwei Farben verschieden?

```
verschieden(rot,gelb).
verschieden(rot,gruen).
...
verschieden(gruen,blau).
```

4. Korrekte Lösung:

```
korrekteFaerbung(L1,L2,L3,L4) :-
    verschieden(L1,L2),
    verschieden(L1,L3),
    verschieden(L2,L3),
    verschieden(L2,L4),
    verschieden(L3,L4).
```

5. Gesamtlösung:

```
?- faerbung(L1,L2,L3,L4), korrekteFaerbung(L1,L2,L3,L4).
L1 = rot
L2 = gelb
L3 = gruen
L4 = rot
```

Wir wollen das obige Beispiel einmal analysieren. Offensichtlich benötigen wir die folgenden Dinge:

- Einschränkung der potentiellen Lösungen (`faerbung`)
- Charakterisierung der korrekten Lösungen
- Gesamtschema:

```
loesung(L) :- moeglicheLoesung(L),korrekteLoesung(L).
```

Dieses Schema wird *generate-and-test* genannt.

Die Komplexität hängt oft von der Menge der möglichen Lösungen ab (hier: $4^4 = 256$ Möglichkeiten).

Um einzusehen, dass die Komplexität immens werden kann, betrachten wir als nächstes Beispiel das Sortieren von Zahlen, `sortiere(UL,SL)`. Hierbei sei `UL` eine Liste von Zahlen und `SL` eine sortierte Variante von `UL`.

1. Wann ist eine Liste sortiert? (korrekte Lösungen)

```
sortiert([]).
sortiert([_]).
sortiert([E1,E2|L]) :- E1 =< E2, sortiert([E2|L]).
```

2. Mögliche Lösungen: `SL` ist eine Permutation von `UL`.

```
perm([],[]).
perm(L1,[E|R2]) :- streiche(E,L1,R1),perm(R1,R2)

streiche(E,[E|R],R).
streiche(E,[A|R],[A|RohneE]) :- streiche(E,R,RohneE).
```

3. Gesamtlösung:

```
sortiere(UL,SL) :-
    perm(UL,SL),  % mögliche Lösung
    sortiert(SL). % korrekte Lösung
```

4. Spezifikation ausführbar:

```
?- sortiere([3,1,4,2,5],SL).
SL = [1,2,3,4,5]
```

Die Komplexität dieses Beispiels liegt für eine n-elementige Liste in der Größenordnung $O(n!)$: Für eine Liste mit zehn Elementen gibt es bereits 3.628.800 mögliche Lösungen...

4.3.2 Musterorientierte Wissensrepräsentation

Ein typisches Beispiel für musterorientiere Wissensrepräsentation ist die Listenverarbeitung. Häufig reicht hier eine einfache Fallunterscheidung, im Fall von Listen auf leere oder nicht-leere Liste. Daraus resultiert ein kleiner, möglicherweise sogar ein-elementiger Suchraum.

Als Beispiel betrachten wir das Prädikat `append(L1,L2,L3)`, das zwei Listen `L1` und `L2` zu einer Liste `L3` konkatenieren soll:

```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```

Dieses Prädikat arbeitet *musterorientiert*: Die Klausel betritt nur leere Listen `L1`, die zweite nur nicht-leere. Bei gegebener Liste `L1` passt also nur eine Klausel - der Suchraum ist ein-elementig.

4.3.3 Verwendung von Relationen

Häufig möchte man Funktionen $f : M_1 \times ... \times M_n \rightarrow M$ implementieren. Die erreicht man in Prolog in Form von Relationen: $f(X_1, ..., X_n, Y) \Leftrightarrow f(X_1, ..., X_n) = Y$.

Einmal implementiert, kann man diese als Funktion gebrauchen:

```
?- f(x1,...,xn,Y).
Y = y
```

Genauso gut kann man sie aber auch Relation bzw. Umkehrfunktion verwenden:

```
?- f(X1,...,Xn,y).
X1 = x1
...
Xn = xn
```

Die Anwendung des obigen `append` als Funktion sieht so aus:

```
?- append([1,2],[3,4],L).
L = [1,2,3,4]
```

Und als Umkehrfunktion lässt sich `append` wie folgt verwenden:

```
?- append(X,[3,4],[1,2,3,4]).
X = [1,2]
```

Wir können es sogar als Umkehrrelation benutzen:

```
?- append(X,Y,[1,2]).
X = []
Y = [1,2];
X = [1]
Y = [2];
X = [1,2]
Y = [];
no
```

Dies lässt sich ausnutzen, um neue Funktionen und Relationen zu definieren:

```
letztes(L,E) :- append(_,[E],L).  % letztes Element einer Liste

member(E,L) :- append(L1,[E|L2],L).  % Element einer Liste

streiche(L1,E,L2)  % zur Berechnung von Permutationen
    :- append(Xs, [E|Ys], L1),
       append(Xs, Ys, L2).

teilliste(T,L) :-  % T als Teilliste in L enthalten?
    append(T1,TL2,L),
    append(T,L2,TL2).
```

Wir merken uns also für die Logikprogrammierung:

- Denke in Relationen (Beziehungen) statt in Funktionen!
- Alle Parameter sind gleichberechtigt (keine Ein-/Ausgabeparameter)!
- Nutze vorhandene Prädikate!

4.4 Programmieren mit Constraints

Die ursprüngliche Motivation für die Einführung der Constraintprogrammierung war die unvollständige Arithmetik:

```
?- X is 5+3.
X = 8

?- 8 is Y+3.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Als Verbesserung wurden spezielle Verfahren zur Lösung arithmetischer (Un-)Gleichungen in das System integriert: Das *Gauß*sche Eliminationsverfahren für Gleichungen, und das Simplexverfahren für Ungleichungen. *Arithmetische Contraints* sind solche Gleichungen und Ungleichungen zwischen arithmetischen Ausdrücken. Es entstand das *Constraint Logic Programming (CLP)*. Dieses ist zwar kein Standard, aber in vielen Systemen realisiert. In Siestus-Prolog oder SWI-Prolog (Version 5.6.x) wird es einfach als Bibliothek dazugeladen:

```
:- use_module(library(clpr)).
```

Alle arithmetischen Constraints werden in geschweifte Klammern eingeschlossen:

```
?- {8 = Y + 3.5}.
Y = 4.5

?- {Y =< 3, 2 + X >= 0, Y - 5 = X}.
X = -2.0,
Y = 3.0
```

Beispiel: Schaltkreisanalyse

Das Ziel ist die Analyse von Spannung und Strom in elektrischen Schaltkreisen. Als erstes müssen wir Schaltkreise als Prolog-Objekte darstellen:

- `wider(R)`: Widerstand vom Wert `R`
- `reihe(S1,S2)`: Reihenschaltung der Schaltkreise `S1` und `S2`
- `parallel(S1,S2)`: Parallelschaltung der Schaltkreise `S1` und `S2`

Die Analyse implementieren wir als Relation `sk(S,V,I)`: Diese steht für einen Schaltkreis `S` mit Spannung `V` und Strom `I`.

```
:- use_module(library(clpr)).

% OHMsches Gesetz
sk(wider(R),V,I) :- {V = I * R}.

% KIRCHHOFFsche Gesetze
sk(reihe(S1,S2),V,I) :-
    {I = I1, I = I2, V = V1 + V2},
    sk(S1,V1,I1),
    sk(S2,V2,I2).

sk(parallel(S1,S2),V,I) :-
    {I = I1 + I2, V = V1, V = V2},
    sk(S1,V1,I1),
    sk(S2,V2,I2).
```

Nun können wir die Stromstärke bei einer Reihenschaltung von Widerständen berechnen:

```
?- sk(reihe(wider(180),wider(470)), 5, I).
I = 0.00769
```

Das System kann uns auch die Relation zwischen Widerstand und Spannung in einer Schaltung ausgeben:

```
?- sk(reihe(wider(R),reihe(wider(R),wider(R))), V, 5).
{ V = 15.0 * R }
```

Beispiel: Hypothekenberechnung

Für die Beschreibung aller nötigen Zusammenhänge beim Rechnen mit Hypotheken verwenden wir folgende Parameter:

- `P`: Kapital
- `T`: Laufzeit in Monaten
- `IR`: monatlicher Zinssatz
- `B`: Restbetrag
- `MP`: monatliche Rückzahlung

Die Zusammenhänge lassen sich nun wie folgt in CLP ausdrücken:

```
mortgage(P,T,IR,B,MP)
  :- {T > 0, T =< 1, B = P * (1 + T * IR) - T * MP}.
mortgage(P,I,IR,B,MP)
  :- {T > 1},
     mortgage(P * (1 + IR) - MP, T - 1, IR, B, MP).
```

Jetzt kann das Prolog-System für uns die monatliche Rückzahlung einer Hypothek bei einer gegebenen Laufzeit ausrechnen:

```
?- mortgage(100000, 180, 0.01, 0, MP).
MP = 1200.17
```

Oder wir fragen das System, wie lange wir eine Hypothek zurückzahlen müssen, wenn wir die monatliche Rückzahlung festhalten:

```
?- mortgage(100000, T, 0.01, 0, 1400).
T = 125.901
```

Es kann uns sogar die Relation zwischen dem aufgenommenen Kapital, der monatlichen Rückzahlung und dem Restbetrag ausgeben:

```
?- mortgage(P, 180, 0.01, B, MP).
{ P = 0.166783 * B + 83.3217 * MP }
```

CLP ist also eine Erweiterung der Logikprogrammierung. Es ersetzt Terme durch Contraint-Strukturen, Datentypen mit festgelegter Bedeutung, und enthält Lösungsalgorithmen für diese Constraints.

Ein konkretes Beispiel ist CLP(R) für die reellen Zahlen:

- Struktur: Terme, reelle Zahlen und arithmetische Funktionen
- Contraints: Gleichungen und Ungleichungen mit arithmetischen Ausdrücken
- Lösungsalgorithmen: Termunifikation, *Gauß*sche Elimination und Simplexverfahren

Weitere Constraint-Strukturen existieren für:

- Boolesche Ausdrücke `(A and (B or C))`: für Hardwareentwurf und -verifikation
- unendliche zyklische Bäume
- Listen
- *endliche Bereiche*, welche zahlreiche Anwendungen in Operations Research finden: für Planungsaufgaben wie zum Beispiel die Maschinenplanung für die Fertigung, die Containerbeladung, die Flughafenabfertigung, und andere

Eine sehr wichtige Constraint-Struktur ist dabei die für endliche Bereiche (finite domains): CLP(FD).

- Struktur: endliche Mengen/Bereiche, dargestellt durch eine endliche Menge ganzer Zahlen
- Elementare Contraints: Gleichungen, Ungleichungen, Elementbeziehungen
- Constraints: logische Verknüpfungen zwischen Constraints
- Lösungsalgorithmen: OR-Methoden zur Konsistenzprüfung

Der Constraint-Löser nimmt nur Konsistenzprüfungen vor: Bei der Gleichung `X = Y` wird zum Beispiel geprüft, ob die Wertebereiche von `X` und `Y` nicht disjunkt sind.

Bei der CLP(FD)-Programmierung geht man also vor wie folgt:

1. definiere den Wertebereich der FD-Variablen
2. beschreibe die Constraints, die diese Variablen erfüllen müssen
3. zähle Werte im Wertebereich auf, d.h. belege FD-Variablen mit ihren konkreten Werten

Der dritte Schritt ist hierbei wegen der Unvollständigkeit des Lösers notwendig. Trotz dieser Unvollständigkeit erhalten wir eine gute Einschränkung des Wertebereichts: So werden die Constraints

```
X in 1..4, Y in 3..6, X = Y
```

zusammengefasst zu `X in 3..4, Y in 3..4`, und die Constraints

```
X in 1..4, Y in 3..6, Z in 4..10, X = Y, Y = Z
```

ergeben `X = 4, Y = 4, Z = 4`.

Die FD-Constraints in Prolog...

- ...sind kein Standard, aber häufig als Bibliotheken vorhanden
- ...können hinzugeladen werden durch

```
:- use_module(library(clpfd)).        % SWI-Prolog
:- use_module(library('clp/bounds')) % Siestus-Prolog
```

- ...enthalten viele elementare Contraints, die mit einem `#` beginnen:
  - `X #= Y` für Gleichheit
  - `X #\= Y` für Ungleichheit
  - `X #> Y` für größer als
  - ...

Beispiel: Krypto-arithmetisches Puzzle

Gesucht ist eine Zuordnung Buchstaben → Ziffern, so dass verschiedene Buchstaben verschiedenen Ziffern entsprechen und die folgende Rechnung stimmt:

```
  SEND
+ MORE
------
 MONEY
```

Mit Siestus finden wir mit folgendem Programm eine Lösung:

```
:- use_module(library(clpfd)).

puzzle(L) :-
    L = [S,E,N,D,M,O,R,Y],
    domain(L,0,9)   % Wertebereich festlegen
    S #> 0, M #> 0  % Constraints festlegen
    all_different(L),
                  1000 * S + 100 * E + 10 * N + D
                + 1000 * M + 100 * O + 10 * R + E
    #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
    labeling([], L).
```

Die Lösung sieht so aus:

```
?- puzzle([S,E,N,D,M,O,R,Y]).
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2
```

Anmerkungen:

- Festlegung bzw. Einschränkung der Wertebereiche:
  - `domain(Vs, Min, Max)`: Wertebereich der Variablen in `Vs` ist $[Min..Max]$
  - `X in Min..Max`: Wertebereich von `X` ist $[Min..Max]$

  Für $[Min..Max]$ sind beliebige Intervallausdrücke erlaubt, also auch Vereininungen, Durchschnitt, u.A.
- auch komplexe *kombinatorische Constraints*, z.B. `all_different(Vs)` für: Variablen in `Vs` haben paarweise verschiedene Werte

- vordefinierte Prädikate zum Werte aufzählen: `labeling([], Vs)` belegt nacheinander die Variablen in `Vs` mit ihren Werten, das erste Argument erlaubt, zusätzliche Steuerungsoptionen anzugeben:

```
?- X in 3..4, Y in 4..5, labeling([], [X,Y]).
X = 3
Y = 4 ;
X = 3
Y = 5 ;
X = 4
Y = 4 ;
X = 4
Y = 5 ;
no

?- X in 4..5, Y in 4..5, Z in 4..5, X #\= Y, Y #\= Z, X #\= Z.
X in 4..5,
Y in 4..5,
Z in 4..5

?- X in 4..5,
   Y in 4..5,
   Z in 4..5,
   X #\= Y,
   Y #\= Z,
   X #\= Z,
   labeling([], [X,Y,Z]).

no
```

Beispiel: 8-Damen-Problem

Das Ziel ist die Platzierung von acht Damen, so dass keine eine andere schlagen kann. Klar ist: In jeder Spalte muss eine Dame sein; die wichtige Frage ist also: In welcher Zeile steht die k-te Dame?

Wie modellieren das Problem wie folgt: Jede Dame entspricht einer FD-Variablen mit einem Wert im Intervall [1..8], der Zeilennummer.

Wir formulieren folgende Constraints:

- Die Damen befinden sich in paarweise verschiedenen Zeilen, da sie sich sonst schlagen können.
- Die Damen dürfen sich auch in den Diagonalen nicht schlagen können.

Wir verallgemeinern das Problem auf ein $N \times N$-Schachbrett.

```
queens(N,L) :-
    length(L,N),      % L ist Liste der Länge N,
                      % d.h. L enthält N verschiedene Variablen
    domain(L, 1, N),  % Wertebereich jeder Dame: [1..N]
    all_safe(L),      % alle Damen sind sicher
    labeling([], L).

all_safe([]).
all_safe([Q|Qs]) :- safe(Q,Qs,1), all_safe(Qs).

safe(_, [], _).
safe(Q, [Q1|Qs], P) :-
    no_attack(Q,Q1,P),
    P1 #= P + 1,
    safe(Q,Qs,P1).

% Damen können waagerecht und diagonal schlagen
no_attack(Q, Q1, P) :-
    Q #\= Q1,
    Q #\= Q1 + P,  % P ist der Spaltenabstand
    Q #\= Q1 - P.
```

Nun liefert uns das Prolog-System für ein 4×4-Schachbrett zum Beispiel zwei Lösungen:

```
?- queens(4,L).
L = [2,4,1,3] ;
L = [3,1,4,2]
```

Eine mögliche Lösung für ein 8×8-Schachbrett lautet:

```
?- queens(8,L).
L = [1,5,8,6,3,7,2,4]
```

Die Berechnung einer Lösung für ein 24×24-Schachbrett benötigt bereits einige Sekunden. Doch das obige Programm lässt sich noch verbesseren: Dazu geben wir Aufzählungsoptionen an.

- keine Option: Die Variablen werden von links nach rechts belegt.

- Option `ff` (first fail): Die Variablen mit dem kleinstem Wertebereich werden zuerst belegt. Dies führt zu weniger Belegungen.

Ersetzen wir die Zeile `labeling([], L)` durch `labeling([ff], L)`, so kann auch noch für ein 100×100-Schachbrett schnell eine Lösung gefunden werden.

4.5 Rechnen in der Logikprogramierung

Rechnen in Prolog entspricht im Wesentlichen dem Beweisen von Aussagen. Aber wie beweist Prolog Aussagen? Um das zu verstehen, betrachten wir zunächst ein einfaches Resolutionsprinzip.

Wir kennen die folgenden Elemente der Logikprogrammierung:

- *Fakten* sind beweisbare Aussagen.
- *Regeln* sind logische Schlussfolgerungen und haben die folgende Semantik: Wenn L `:-` $L_1, ..., L_n$`.` eine Regel ist, und $L_1, ..., L_n$ beweisbar sind, dann ist auch L beweisbar. Diese Regel wird als *modus ponens* oder auch *Abtrennungsregel* bezeichnet.
- *Anfragen* sind zu überprüfende Aussagen mit der folgenden Semantik: Wenn `?-` $L_1, ..., L_n$`.` eine Anfrage ist, dann wird überprüft, ob $L_1, ..., L_n$ beweisbar ist mit den gegebenen Fakten und Regeln.

Dies führt zu der folgenden Idee: Um die Aussage einer Anfrage zu überprüfen, suche die dazu passenden Regeln und kehre den modus ponens um zum:

Einfaches Resolutionsprinzip: Reduziere den Beweis der Aussage L auf den Beweis der Aussagen $L_1, ..., L_n$, falls L `:-` $L_1, ..., L_n$ Regel ist. Hier sind Fakten Regeln mit leerem Rumpf.

Wir betrachten das folgende Beispiel:

```
ehemann(monika, herbert).
mutter(susanne, monika).
vater(susanne, herbert) :-
    ehemann(monika, herbert),
    mutter(susanne, monika).

?- vater(susanne, herbert).
 |- Regel von vater
?- ehemann(monika, herbert), mutter(susanne, monika).
 |- Faktum
?- mutter(susanne, monika).
 |- Faktum
```

```
?- .
```

Kann eine Anfrage in mehreren Schritten mittels des Resolutionsprinzips auf die leere Anfrage reduziert werden, dann ist die Anfrage beweisbar.

Das Problem ist, dass die Regeln oft nicht „direkt“ passen: So passt zum Beispiel `vater(K,V) :- ehemann(M,V), mutter(K,M).` nicht zu `?- vater(susanne, herbert).` Aber `K` und `V` sind Variablen, wir können also beliebige Objekte einsetzen. Ersetze also `K` durch `susanne` und `V` durch `herbert`.

Eine *Substitution* ist eine Abbildung $\sigma : Terme \to Terme$, die Variablen durch Terme ersetzt, mit folgenden Eigenschaften:

1. Für alle Terme $f(t_1, ..., t_n)$ gilt: $\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$. σ ist strukturerhaltend oder Homomorphismus. Diese Eigenschaft gilt analog für Literale.

2. Die Menge $\{X \mid X \text{ Variable mit } \sigma(X) \neq X\}$ ist endlich.

Daher ist σ eindeutig darstellbar als Paarmenge $\{X \mapsto \sigma(X) \mid X \neq \sigma(X), X \text{ Variable}\}$. In unserem Beispiel sieht σ also so aus: $\sigma = K \mapsto susanne, V \mapsto herbert$. Die Anwendung der Substitution sieht dann so aus: $\sigma(vater(K, V)) = vater(susanne, herbert)$.

Aber wir müssen auch die Variablen in Anfragen wie `?- ehemann(monika, M).` ersetzen können. Dazu Bedarf es der *Unifikation*: Terme müssen durch Variablenersetzung gleich gemacht werden. Im Beispiel `datum(Tag, Monat, 83), datum(3, M, J)` gibt es mehrere mögliche Substitutionen:

$$\sigma_1 = Tag \mapsto 3, Monat \mapsto 4, M \mapsto 4, J \mapsto 83.$$

$$\sigma_2 = Tag \mapsto 3, Monat \mapsto M, J \mapsto 83.$$

Sowohl σ_1 als auch σ_2 machen Terme gleich, σ_1 ist aber spezieller.

Eine Substitution σ heißt *Unifikator* für t_1 unt t_2, falls $\sigma(t_1) = \sigma(t_2)$. t_1 und t_2 heißen dann *unifizierbar*.

σ heißt *allgemeinster Unifikator (most general unifier (mgu))*, falls für alle Unifikatoren σ' eine Substitution ϕ exisitiert mit $\sigma' = \phi \circ \sigma$ (hier: $\phi \circ \sigma(t) = \phi(\sigma(t))$).

Es ist wichtig, mit mgus zu rechnen: Wir erhalten weniger Beweise, und müssen also weniger suchen. Es stellt sich also die Frage: Gibt es immer mgus? Und wie kann man diese berechnen?

Die Antwort hat *Robinson* im Jahr 1965 gefunden: Ja, es gibt immer mgus. Für ihre Berechnung definieren wir den Begriff der *Unstimmigkeitsmengen von Termen*:

Sind t, t' Terme, dann ist die *Unstimmigkeitsmenge (disagreement set)* $ds(t, t')$ definiert durch:

1. Falls $t = t'$: $ds(t, t') = \emptyset$
2. Falls t oder t' Variable und $t \neq t'$: $ds(t, t') = \{t, t'\}$
3. Falls $t = f(t_1, ..., t_n)$ und $t' = g(s_1, ..., s_m)$:
   - Falls $f \neq g$ oder $m \neq n$: $ds(t, t') = \{t, t'\}$
   - Falls $f = g$ und $m = n$ und $(\forall i < k : t_i = s_i)$ und $t_k \neq s_k$: $ds(t, t') = ds(t_k, s_k)$

Intuitiv kann man diese Fallunterscheidung so begreifen: $ds(t, t')$ enthält die Teilterme von t und t' an der linkesten Position, an denen t und t' verschieden sind.

Daraus ergibt sich unmittelbar der folgende *Unifikationsalgorithmus*.

Unifikationsalgorithmus:

Eingabe: Terme (Literale) t_0, t_1
Ausgabe: mgu σ für t_0, t_1, falls diese unifizierbar, und „fail“ sonst

1. `k := 0;` σ_0 `:= {}`
2. Falls $\sigma_k(t_0) = \sigma_k(t_1)$, dann ist σ_k mgu
3. Falls $ds(\sigma_k(t_0), \sigma_k(t_1)) = \{x, t\}$ mit x Variable und x kommt nicht in t vor, dann: σ_{k+1} `:=` $x \mapsto t \circ \sigma_k$; `k := k + 1`; gehe zu 2; sonst: „fail“

Wir wollen den Algorithmus einmal an einem Beispiel nachvollziehen:

```
t0 = ehemann(monika, M)
t1 = ehemann(F, herbert)

ds(t0, t1) = {F, monika}
    => sigma1 = {F |-> monika}
ds(sigma1(t0), sigma(t1)) = {M, herbert}
    => sigma2 = {M |-> herbert, F |-> monika}

sigma2(t0) = sigma2(t1)
    => sigma2 ist mgu
```

Ein zweites Beispiel soll zeigen, wie die Unifikation auch fehlschlagen kann:

```
t0 = equ(f(1), g(X))
t1 = equ(Y,Y)

ds(t0, t1) = {Y, f(1)}
    => sigma1 = { Y |-> f(1)}
ds(sigma1(t0), sigma1(t1)) = {g(X), f(1)}
    => "fail"
```

Ein letztes Beispiel soll den Sinn der Überprüfung von Schritt 3 zeigen, ob x nicht in t vorkommt:

```
t0 = X
t1 = f(X)

ds(t0, t1) = {X, f(X)}
    => "fail"
```

Der *Vorkommenstest (occurcheck)* ist also relevant. Viele Prolog-Systeme verzichten aber aus Gründen der Effizienz auf diesen Test, da er selten erfolgreich ist. Theoretisch kann dies natürlich zu einer fehlerhaften Unifikation und damit zur Erzeugung zyklischer Terme führen.

Es gilt folgender *Satz (Unifikationssatz von Robinson):* Seien t_0, t_1 Terme. Sind diese unifizierbar, dann gibt der obige Algorithmus einen mgu für t_0, t_1 aus. Sind sie es nicht, dann gibt er „fail“ aus.

Wir schließen daraus: Unifizierbare Terme haben immer einen mgu!

Daraus erhalten wir unmittelbar das *Allgemeine Resolutionsprinzip*, es vereint Resolution und Unifikation:

Der Beweis der Anfrage `?-` $A_1, ..., A_{i-1}, A_i, A_{i+1}, ..., A_n$`.` kann reduziert werden zum Beweis der Anfrage `?-` $\sigma(A_1, ..., A_{i-1}, L_1, ..., L_m, A_{i+1}, ..., A_n)$`.` falls `L :-` $L_1, ..., L_m$`.` eine Regel ist (mit neuen Variablen) und σ ein mgu für A_i und `L`.

Nun bleiben nur noch folgende Punkte offen:

1. Welches Literal A_i wird bewiesen? In Prolog wird immer das linkeste Literal gewählt: $i = 1$.

2. Welche Regel (Welches Faktum) wird gewählt? Die sichere Methode wäre, alle Regeln gleichzeitig zu wählen; diese Methode bringt natürlich den Nachteil mit sich, dass sie sehr aufwändig ist. In Prolog verzichtet man also auf die Sicherheit zugunsten der Effizienz. Dort verwendet man die folgende *Backtracking-Methode*:

   (a) Die Klauseln haben eine Reihenfolge, und zwar die, in der sie im Programm definiert werden.

(b) In einem Resolutionsschritt wird die *erste passende* Klausel für das linke Literal gewählt. Bei Sackgassen werden die letzen Schritte rückgängig gemacht und die nächste Alternative probiert.

(c) Bei der Anwendung einer Regel werden die Variablen durch Unifikation durch Terme ersetzt. Dann wird eine Variable an einen Term *gebunden* oder auch *instantiiert*.

Wir veranschaulichen die *Auswerungsstrategie von Prolog* an folgendem Beispiel:

```
p(a).
p(b).
q(b).

?- p(X), q(X).
    |- { X |-> a}
?- q(a).
    Sackgasse. Rücksetzen:
    |- { X |-> b}
?- q(b).
    |- {}
?- .
```

Folgendes Programm bringt dadurch jedoch Probleme mit sich:

```
p :- p.
p.

?- p.
    |- ?- p.
    |- ?- p.
    ...
```

Das System landet in einer Endlosschleife, statt `yes` zurückzugeben. Prolog ist also unvollständig als „Theorembeweiser“.

Wir betrachten ein weiteres Beispiel für die Relevanz der Klauselreihenfolge:

```
last([K|R], E) :- last(R,E).
last([E], E).
```

```
?- last(L, 3).
    |- { L |-> [K1|R1]}
?- last(R1, 3).
    |- { R1 |-> [K2|R2]}
?- last(R2, 3).
...
```

Man kann sich also folgende Empfehlung merken: Klauseln für Spezialfälle sollten stets *vor* allgemeinere Klauseln angegeben werden! Denn vertauschen wir im letzten Beispiel die Klauseln, so terminiert die Anfrage sofort:

```
?- last(L, 3).
    |- { L |-> [3]}
?- .
```

4.6 Der „Cut“-Operator

Mit „Cut“ (`!`) kann man das Backtracking teilweise unterdrücken. Die möchte man manchmal aus verschiedenen Gründen:

1. Effizienz (Speicherplatz und Laufzeit)
2. Kennzeichnen von Funktionen
3. Verhinderung von Laufzeitfehlern (z.B. bei `is`)

In Prolog kann `!` anstelle von Literalen im Regelrumpf stehen: `p :- q, !, r.` Operational bedeutet dies: Wird diese Regel zum Beweis von `p` benutzt, dann gilt:

1. Falls `q` nicht beweisbar: wähle nächste Regel für `p`.
2. Falls `q` beweisbar: `p` nur beweisbar, falls `r` beweisbar. Mit anderen Worten, es wird kein ein Alternativbeweis für `q` und keine andere Regel für `p` ausprobiert.

Wir werfen einen Blick auf das folgende Beispiel:

```
ja :- ab(X), !, X = b.
ja.
```

```
ab(a).
ab(b).

?- ja.
```

Rein logisch ist die gemachte Anfrage auf zwei Arten beweisbar. Operational wird jedoch die erste Regel angewandt, `X` an `a` gebunden, es folgt ein Cut, dann ein Fehlschlag weil `X = b` nicht bewiesen werden kann, und schließlich wird keine Alternative mehr ausprobiert. Also Vorsicht mit Cut!

Häufig verwendet man einen Cut zur Fallunterscheidung:

```
p :- q, !, r.
p :- s.
```

Dies entspricht so etwas wie

```
P :- if q then r else s.
```

Tatsächlich gibt es genau dafür eine spezielle Syntax in Prolog: `p :- q -> r; s.`

Als Beispiel wollen wir einmal die Maximumrelation implementieren:

```
max(X,Y,Z) :- X >= Y, !, Z = X.
max(X,Y,Z) :- Z = Y.  % rein logisch unsinnig!
```

Alternativ könnte man diese auch so aufschreiben:

```
max(X,Y,Z) :- X >= Y, -> Z = X; Z = Y.
```

4.7 Negation

Das folgende Beispiel soll zeigen, warum man häufig die Negation benötigt:

```
geschwister(S, P) :-  mutter(S, M), mutter(P, M).
```

Hier fehlt allerdings noch eine Bedingung wie `nicht S = P`, sonst wäre jeder, der eine Mutter hat, sein eigenes Geschwister.

In Prolog ist die *Negation als Fehlschlag (NAF)* implementiert: `\+ p` ist beweisbar, falls alle Beweise für `p` fehlschlagen.

```
?- \+ monika = susanne.
yes
```

Betrachtet man aber das Beispiel `p :- \+ p.`, so ergibt sich bei logischer Negation: $\neg p \Rightarrow p \equiv \neg(\neg p) \vee p \equiv p \vee p \equiv p$. In Prolog landen wir aber in einer Endlosschleife. *Clark* hat im Jahr 1978 deshalb die NAF als *negation as finite failure* eingeführt: Falls alle Beweise für `p` endlich und fehlgeschlagen sind, dann ist `\+ p` beweisbar. Diese ist effektiv implementierbar: Zum Beispiel können wir `p :- \+ q.` formulieren als:

```
p :- q, !, fail.
p.
```

Es gibt allerdings noch ein weiteres Problem: Die Negation in Prolog ist inkorrekt, falls das negierte Literal Variablen enthält.

```
p(a,a).
p(a,b).

?- \+ p(b,b).
yes
?- \+ p(X,b).
no (statt {X |-> b}, wegen Fehlschlag werden Variablen nie gebunden!)
```

Die Konsequenz daraus ist: Beim Beweis von `\+ p` darf `p` keine Variablen enthalten! Zum Schluss werfen wir noch einen letzten Blick auf unser Verwandschaftsbeispiel:

```
geschwister(S,P) :-
    mutter(S, M),
    mutter(P, M),
    \+ S = P.  % hier ok, da S und P immer gebunden sind
```

Es ist also zu empfehlen, negierte Literale in den Regeln möglichst weit rechts anzugeben!

Abbildungsverzeichnis

Index