Kapitel 10

Fallstudien

Wir betrachten jetzt das Zusammenspiel von Algorithmen und *Datenstrukturen* am Beispiel von drei anspruchsvolleren Programmierpoblemen.

Jede Programmiersprache stellt einige Datenstrukturen direkt zur Verfügung. Bei Standard ML sind dies unter anderem Tupel, Listen und Bäume. Datenstrukturen, die in einer Programmiersprache nicht direkt verfügbar sind, werden mithilfe der verfügbaren Datenstrukturen implementiert. Komplexere Datenstrukturen sind fast immer mit Darstellungsinvarianten verbunden. Wir haben das am Beispiel der effizienten Realisierung von Schlangen in Abschnitt 9.3 gesehen. Dieses Beispiel zeigt auch, dass die Entwicklung effizienter Algorithmen mit der Entwicklung effizienter Datenstrukturen Hand in Hand geht.

Zunächst zeigen wir, wie man mithilfe sogenannter Rot-Schwarz-Bäume endliche Mengen und Funktionen effizient realisieren kann. Das Beispiel zeigt, dass die Entwicklung von Algorithmen und Datenstrukturen eine kreative Angelegenheit ist, die zu außerordentlich eleganten Konstruktionen führen kann. Dazu kommt, dass die am Beispiel von Rot-Schwarz-Bäumen vorgeführten Techniken für viele Anwendungen wichtig sind.

Die zweite Fallstudie zeigt, wie man große Zahlen implementiert. Dabei lernen wir grundlegende Dinge über Zahldarstellungen und programmieren die aus der Schule bekannten Algorithmen für Addition, Subtraktion, Multiplikation und Division.

Zuletzt betrachten wir das Damenproblem. Dieses Problem lösen wir mithilfe eines Suchalgorithmus, den wir abstrakt mit Transitionsregeln beschreiben.

10.1 Binäre Suche

Binäre Suche ist eine Technik für die effiziente Realisierung von endlichen Mengen und Funktionen. Binäre Suche setzt eine geordnete Menge voraus. Ein typisches Beispiel für eine geordnete Menge sind die ganzen Zahlen. In Abschnitt 9.2 haben wir eine Typabstraktion

```
type set
val  empty  : set
val  insert : int * set -> set
val  member : int * set -> bool
```

für endliche Mengen über `int` eingeführt. Mit binärer Suche lassen sich die Operationen `insert` und `member` so realisieren, dass sie höchstens logarithmische Laufzeit haben (bezüglich der Größe der Menge, auf die sie angewendet werden). Das stellt eine erhebliche Verbesserung gegenüber der naiven Implementierung in Abschnitt 9.2 dar, bei der `member` mindestens lineare Laufzeit hat.

Die Grundidee hinter binärer Suche ist einfach. Sie stellt nichtleere, endlichen Menge $X \subseteq \mathbb{Z}$ durch Tripel

$$(L, x, R)$$

wie folgt dar:

1. *Darstellungseigenschaft*: $X = L \cup \{x\} \cup R$.
2. *Ordnungseigenschaft*: $\forall y \in L\ \forall z \in R\colon\ y < x < z$.
3. *Gleichgewichtsseigenschaft*: L und R sind ungefähr gleich groß.

Um zu entscheiden, ob für ein $y \in \mathbb{Z}$ die Beziehung $y \in X$ gilt, kann man dann wie folgt vorgehen:

1. Wenn $y = x$, dann $y \in X$.
2. Wenn $y < x$, dann $y \in X$ genau dann, wenn $y \in L$.
3. Wenn $x < y$, dann $y \in X$ genau dann, wenn $y \in R$.

Dabei ist der entscheidende Punkt, dass die Mengen L und R nur noch etwa halb so groß sind wie X. Wenn die Tripeldarstellung rekursiv auch für L und R vorliegt, kann $y \in X$ also mit der Laufzeit $O(\log |X|)$ getestet werden.

10.1.1 Suchbäume

Binäre Suche stellt Mengen über $\mathbb{Z}$ also durch binäre Bäume der Form

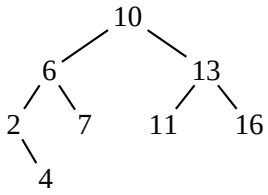


Abbildung 10.1: Eine Baumdarstellung der Menge {2, 4, 6, 7, 10, 11, 13, 16}

```
datatype stree = E | N of stree * int * stree
```

dar. Der Konstruktor `E` repräsentiert die leere Menge. Der Konstruktor `N` ist für nichtleere Mengen in Tripeldarstellung zuständig. Abbildung 10.1 zeigt ein Beispiel für eine Baumdarstellung einer Menge. Dabei haben haben wir alle Knoten weggelassen, die die leere Menge repräsentieren (Konstruktor `E`).

Wir nennen die Werte des Typs `stree` *S-Bäume*. Die *Infixprojektion* von S-Bäumen definieren wir wie folgt:

```
fun pro E          = []
  | pro (N(a,x,b)) = pro a @ [x] @ pro b
val pro : stree -> int list
```

Ein S-Baum stellt die durch seine Infixprojektion beschriebene Menge dar.

Ein S-Baum heißt *Suchbaum* genau dann, wenn seine Infixprojektion eine aufsteigend sortierte Liste ohne Doppelauftreten ist. Suchbäume sind genau die S-Bäume, die die für die Tripeldarstellung erforderliche Ordnungseigenschaft einhalten. Der in Abbildung 10.1 gezeigte S-Baum ist ein Suchbaum.

Die Prozedur

```
fun member (y, E)        = false
  | member (y, N(a,x,b)) = case Int.compare(y,x) of
                             LESS    => member(y,a)
                           | EQUAL   => true
                           | GREATER => member(y,b)
val member : int * stree -> bool
```

testet, ob eine Zahl in einer durch einen Suchbaum dargestellten Menge liegt. Für Suchbäume, die die Gleichgewichtseigenschaft haben (siehe Definition der Tripeldarstellung), hat `member` höchstens logarithmische Laufzeit.

Wir wenden uns jetzt der Realisierung der *Einfügeoperation* `insert` zu. Zunächst betrachten wir eine naive Realisierung der Einfügeoperation, die zu Suchbäumen Suchbäume liefert, sich aber nicht um die Gleichgewichtseigenschaft

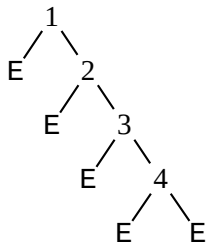
kümmert:

```
fun insert (y, E)        = N(E,y,E)
  | insert (y, N(a,x,b)) = case Int.compare(y,x) of
                             LESS    => N(insert(y,a), x, b)
                           | EQUAL   => N(a,y,b)
                           | GREATER => N(a, x, insert(y,b))
val insert : int * stree -> stree
```

Nehmen Sie an, dass wir, ausgehend von E, die Elemente

1, 2, 3, 4

mit dem obigen Prozedur einfügen (in der Reihenfolge von links nach rechts). Dann bekommen wir den listenartigen Suchbaum



der die Gleichgewichtsbedingung eklatant verletzt.

10.1.2 Rot-Schwarz-Bäume

Wir definieren jetzt die Gleichgewichtsbedingung präzise. Wir tun das so, dass `insert` mit kleinen Kosten feststellen kann, ob die Gleichgewichtsbedingung durch das Einfügen eines neuen Knotens verletzt wird und eine eventuelle Verletzung mit kleinen Kosten reparieren kann. Dazu färben wir jeden Knoten eines S-Baums mit einer der Farben rot und schwarz. Knoten werden durch die Konstruktoren E und N eingeführt. Entspechend sprechen wir von *E-Knoten* und *N-Knoten*. E-Knoten werden immer schwarz eingefärbt. S-Bäume mit farbigen Knoten nennen wir *C-Bäume* und stellen sie wie folgt dar:

```
datatype color = R | B
datatype ctree = E | N of color * ctree * int * ctree
```

Ein C-Baum heißt *rot-balanciert* genau dann, wenn keiner seiner Pfade zwei aufeinanderfolgende rote Knoten enthält. Ein *maximaler Pfad* eines Baums ist ein Pfad von der Wurzel zu einem Blatt. Ein C-Baum heißt *schwarz-balanciert* genau dann, wenn seine maximalen Pfade alle die gleiche Anzahl schwarzer Knoten

enthalten. Ein C-Baum heißt *RB-Baum* genau dann, wenn er rot- und schwarzbalanciert ist. In einem RB-Baum unterscheiden sich die Längen von zwei maximalen Pfaden höchstens um den Faktor 2. Offensichtlich ist jeder Unterbaum eines RB-Baums wieder ein RB-Baum.

Die Tiefe eines Baums ist die Länge der längsten maximalen Pfade. Um zu zeigen, dass die Tiefe eines RB-Baums logarithmisch durch die Anzahl seiner N-Knoten nach oben beschränkt ist, benötigen wir einige Hilfsdefinitionen. Die logarithmische Beschränkung der Tiefe garantiert die logarithmische Laufzeit von `member`.

Die *N-Größe* $n(b)$ eines C-Baums b ist die Anzahl der N-Knoten von b. Die *S-Höhe* $s(b)$ eines schwarz-balancierten C-Baums b ist die Anzahl der schwarzen Knoten auf einem maximalen Pfad. Für jeden schwarz-balancierten C-Baum b definieren wir $s'(b)$ wie folgt:

$$s'(b) = \text{if Wurzel von } b \text{ rot then } s(b) \text{ else } s(b) - 1$$

Proposition 10.1.1 *Für die Tiefe $t(b)$ jedes RB-Baums b gilt: $t(b) \leq 2s(b)$.*

Proposition 10.1.2 *Für jeden RB-Baum b gilt: $n(b) \geq 2^{s'(b)} - 1$.*

Beweis Durch Induktion über $g(b)$.

Sei $g(b) = 1$. Dann $b = E$, also $n(b) = 0$ und $s'(b) = 0$. Also $n(b) \geq 2^{s'(b)} - 1$.

Sei $g(b) > 1$. Dann $b = N(c, b_1, x, b_2)$. Wenn die Wurzel von b rot ist, dann sind die Wurzeln von b_1 und b_2 beide schwarz und folglich $s'(b_1) = s'(b_2) = s'(b) - 1$. Wenn die Wurzel von b schwarz ist, dann $s'(b_1), s'(b_2) \geq s'(b) - 1$. Also gilt in jedem Fall $s'(b_1), s'(b_2) \geq s'(b) - 1$. Folglich

$$\begin{aligned} g(b) &= 1 + g(b_1) + g(b_2) \\ &\geq 2^{s'(b_1)} + 2^{s'(b_2)} - 1 && \text{Induktionsannahme} \\ &\geq 2^{s'(b)-1} + 2^{s'(b)-1} - 1 \\ &= 2^{s'(b)} - 1 \end{aligned}$$

□

Proposition 10.1.3 *Für jeden RB-Baum b gilt: $t(b) \leq 2 + 2\log_2(n(b) + 1)$.*

Beweis Folgt sofort aus den Propositionen 10.1.1 und 10.1.2. □

Damit ist sichergestellt, dass `member` für RB-Bäume höchstens logarithmische Laufzeit hat.

Ein RB-Baum heißt *Rot-Schwarz-Baum* genau dann, wenn er als S-Baum betrachtet ein Suchbaum ist. Offensichtlich ist jeder Unterbaum eines Rot-Schwarz-Baums wieder ein Rot-Schwarz-Baum. Wir werden gleich sehen, dass `insert` für Rot-Schwarz-Bäume mit logarithmischer Laufzeit realisiert werden kann. Rot-Schwarz-Bäume sind also die angestrebte effiziente Darstellung für Mengen. Mengen werden also als C-Bäume dargestellt, die die Darstellungsinvariante erfüllen, Rot-Schwarz-Bäume zu sein.

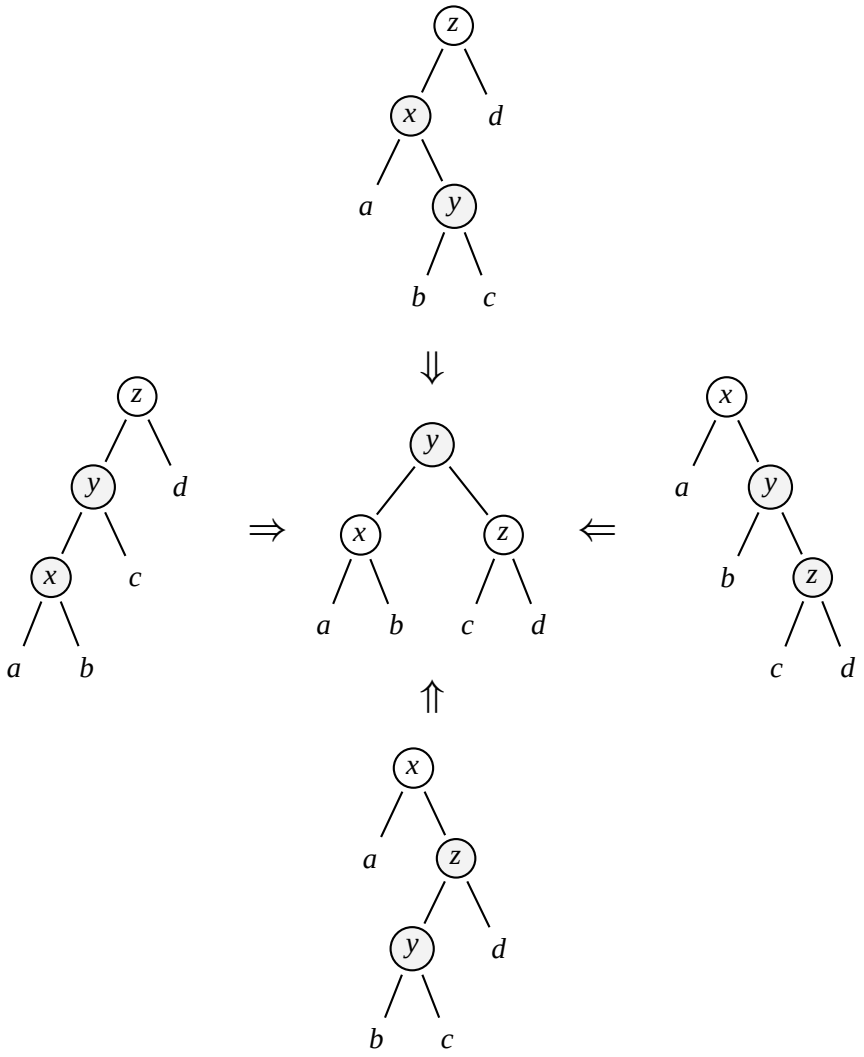
Die naive Einfügeoperation fügt höchstens einen neuen N-Knoten hinzu. Wir legen fest, dass neu eingefügte N-Knoten zunächst immer die Farbe rot bekommen. Die naive Einfügeoperation kann also höchstens die Rot-Balanciertheit eines Rot-Schwarz-Baums zerstören. Falls dies der Fall ist, gibt es im Baum genau zwei aufeinanderfolgende rote Knoten, wobei der untere der neu eingefügte N-Knoten ist. Wir sprechen von einem *Rot-Konflikt*. Rot-Konflikte, deren oberer Knoten die Wurzel ist, kann man durch Schwarzfärben der Wurzel eliminieren (Schwarzfärben der Wurzel überführt jeden RB-Baum in einen RB-Baum).

Abbildung 10.2 zeigt die sogenannten *Rotationsregeln* für Rot-Schwarz-Bäume. Auf jeden Rot-Schwarz-Baum mit einer schwarzen Wurzel, der einen Rot-Konflikt direkt unterhalb der Wurzel enthält, ist eine der Rotationsregeln anwendbar. Die Regeln eliminieren den Rot-Konflikt und liefern einen Baum mit roter Wurzel. Man überzeugt sich leicht, dass die Regeln Rot-Schwarz-Bäume in Rot-Schwarz-Bäume überführen und dabei die S-Höhe und die dargestellte Menge unverändert erhalten.

Durch die Anwendung einer Rotationsregel auf einen Teilbaum kann man einen tieferliegenden Rot-Konflikt entweder eliminieren oder zumindest weiter nach oben schieben. Durch wiederholtes Anwenden der Rotationsregeln und durch Schwarzfärben der Wurzel sind wir also in der Lage, jeden Rot-Konflikt zu eliminieren.

Mit den geschilderten Ideen können wir die Einfügeprozedur in Abbildung 10.3 schreiben, die zu Rot-Schwarz-Bäumen Rot-Schwarz-Bäume liefert. Die Prozeduren `rotate`, `rotateL` und `rotateR` realisieren die Rotationsregeln. Die Prozedur `insert'` liefert einen schwarz-balancierten Suchbaum, der nur dann einen Rot-Konflikt enthält, wenn dieser die Wurzel einschließt. Die Prozedur `insert` sorgt in jedem Fall dafür, dass die Wurzel schwarz gefärbt wird.[1] Damit wird die Rot-Balanciertheit wieder hergestellt wird, falls `insert'` einen Baum mit einem Rot-Konflikt liefert.

[1] Da `insert'` den trivialen Baum `E` nicht liefern kann, ist die zweite Regel `E => E` der Fallunterscheidung von `insert` eigentlich unnötig. Wir haben sie eingeführt, um eine nicht erschöpfende Fallunterscheidung zu vermeiden.



Rote Knoten sind grau schattiert

Abbildung 10.2: Die Rotationsregeln für Rot-Schwarz-Bäume

```
datatype color = R | B
datatype ctree = E | N of color * ctree * int * ctree

fun rotate (x,y,z,a,b,c,d) = N(R, N(B,a,x,b), y, N(B,c,z,d))

fun rotateL (B, N(R,N(R,a,x,b),y,c), z, d) = rotate(x,y,z,a,b,c,d)
  | rotateL (B, N(R,a,x,N(R,b,y,c)), z, d) = rotate(x,y,z,a,b,c,d)
  | rotateL         body                   = N body

fun rotateR (B, a, x, N(R,b,y,N(R,c,z,d))) = rotate(x,y,z,a,b,c,d)
  | rotateR (B, a, x, N(R,N(R,b,y,c),z,d)) = rotate(x,y,z,a,b,c,d)
  | rotateR         body                   = N body

fun insert' (y, E)              = N(R,E,y,E)
  | insert' (y, N(color,a,x,b)) =
      case Int.compare(y,x) of
        LESS    => rotateL(color, insert'(y,a), x, b)
      | EQUAL   => N(color,a,y,b)
      | GREATER => rotateR(color, a, x, insert'(y,b))

fun insert (y,a) = case insert'(y,a) of
                     N(_,a,x,b) => N(B,a,x,b)
                   | E          => E
```

Abbildung 10.3: Einfügeprozedur für Rot-Schwarz-Bäume

Wir haben jetzt eine effiziente Implementierung für endliche Mengen über `int`. Mithilfe eines Funktors können wir die Festlegung auf `int` eliminieren und endliche Mengen für beliebige Typen implementieren, für die eine Vergleichsprozedur `compare` verfügbar ist (siehe Abschnitt 9.4). Es sind nur wenige Modifikationen erforderlich, um die für Anwendungen wichtige Typabstraktion "endliche Funktion" mithilfe von Rot-Schwarz-Bäumen effizient zu implementieren (siehe Aufgaben 9.2 und 10.3).

In der Praxis ist es oft sinnvoll, die Typabstraktion "endliche Menge" um eine Operation

```
val delete : int * set -> set
```

zu erweitern, die zu x und M die Menge $M - \{x\}$ liefert. Mithilfe von Rot-Schwarz-Bäumen kann man `delete` mit logarithmischer Laufzeit realisieren (siehe [**?**]).

Sortieren mit Rot-Schwarz-Bäumen

Mit Rot-Schwarz-Bäume kann man übrigens Listen ohne Doppelauftreten mit der Laufzeit $O(n \log n)$ sortieren (n ist die Länge der Listen):

```
fun pro E            xs = xs
  | pro (N(_,a,x,b)) xs = pro b (x::pro a xs)

val pro : ctree -> int list -> int list

fun sort xs = rev (pro (foldl insert E xs) nil)

val sort : int list -> int list
```

Wenn man `sort` auf eine Liste mit Doppelauftreten anwendet, bekommt man die sortierte Liste, in der alle Doppelauftreten eliminiert sind. Dabei berechnet `pro` die reversierte Infixprojektion eines C-Baums. Die Laufzeit $O(n \log n)$ kann man wie folgt einsehen: `rev`, `pro` und `foldl` benötigen jeweils lineare Laufzeit, und `insert` benötigt höchstens logarithmische Laufzeit. Da `insert` für eine Liste der Länge n gerade n-mal aufgerufen wird, benötigen alle Aufrufe von `insert` nur die Laufzeit $O(n \log n)$.

10.2 Große Zahlen

Wir haben bereits festgestellt, dass Moscow ML die Werte des Typs `int` auf relativ kleine ganze Zahlen beschränkt. Diese Beschränkung macht es möglich, die arithmetischen Operationen direkt durch entsprechende Hardware-Operationen zu realisieren. Wir überlegen uns jetzt, wie man große Zahlen mithilfe von Listen und den Zahlen in `int` realisieren kann. Beispielweise haben wir die Möglichkeit, die Zahl

987654321123456789

gemäß ihrer Dezimaldarstellung durch die Liste

[9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9]

darzustellen. Die Dezimaldarstellung stellt Zahlen zur Basis 10 dar. Es besteht die Möglichkeit, Zahlen bezüglich einer beliebigen Basis $B \geq 2$ darzustellen. Wenn wir große Zahlen bezüglich einer größeren Basis darstellen, bekommen wir kürzere Listen und schnellere arithmetische Operationen, da wir die Hardware-Operationen besser ausnützen können. Wenn wir die obige Zahl zur Basis 1000 darstellen, bekommen wir die Liste

[987, 654, 321, 123, 456, 789]

10.2.1 Zahldarstellungen

Die folgende Proposition stellt die Grundlage für die Darstellung von Zahlen bezüglich einer *Basis* $B \geq 2$ bereit.

Proposition 10.2.1 (Zahldarstellungen) *Seien $B, n \in \mathbb{N}$ mit $B \geq 2$. Dann existieren zu jeder Zahl*

$$x \in \{0, \ldots, B^{n+1} - 1\}$$

eindeutig bestimmte Zahlen $a_0, \ldots, a_n \in \{0, \ldots, B-1\}$ mit

$$x = a_n \cdot B^n + \cdots + a_0 \cdot B^0 = \sum_{i=0}^{n} a_i \cdot B^i$$

Die Zahlen 0 bis $B-1$ bezeichnet man als die *Ziffern* der Zahldarstellung zur Basis B.

Dezimaldarstellung

Um die Proposition zu verstehen, betrachten wir den Spezialfall $B = 10$, der der üblichen *Dezimaldarstellung* von Zahlen entspricht. Beispielsweise gilt

$$6701 = 6 \cdot 10^3 + 7 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0$$

Die Koeffizienten a_i der Proposition sind also die Ziffern, mit denen eine Zahl dargestellt wird.

Gegeben eine Basis, können wir Zahlen durch die Liste ihrer Ziffern darstellen. Gegeben die Basis 10, kann die Zahl 6701 durch die Liste $[6, 7, 0, 1]$ dargestellt werden. Wenn wir darauf bestehen, dass die Listen keine führenden Nullen enthalten, wird jede natürlich Zahl durch genau eine Liste dargestellt. Die Zahl 0 wird dann durch die leere Liste dargestellt.

Binärdarstellung und Hardware-Arithmetik

Die Darstellung von Zahlen zur Basis 2 heißt *Binärdarstellung*. Als Ziffern benötigt man bei der Binärdarstellung nur `0` und `1`. Die Hardware von Computern rechnet mit den Binärdarstellungen von Zahlen. Dabei werden alle Zahlen durch Tupel einer festen Stelligkeit dargestellt, der sogenannten *Darstellungsbreite*. Heutige Computer (zum Beispiel PCs) arbeiten meistens mit einer *Darstellungsbreite* von 32, man spricht von *32-Bit Computern*.

Mit einer Darstellungsbreite von 32 kann man genau 2^{32} verschiedene Zahlen binär darstellen. Da man auch negative Zahlen benötigt, stellt man mit 32 Bit den Bereich

$$\{-2^{31}, \ldots, 2^{31} - 1\}$$

dar. Moscow ML stellt auf PCs allerdings nur den Bereich

$$\{-2^{30}, \ldots, 2^{30} - 1\}$$

zur Verfügung, da ein Bit für Verwaltungszwecke benötigt wird (automatische Speicherbereinigung, siehe Abschnitt 15.3).

10.2.2 Additionsalgorithmus

In der Schule haben Sie gelernt, wie man mit Dezimaldarstellungen rechnet. Die dabei verwendeten Algorithmen lassen sich problemlos auf Darstellungen bezüglich anderer Basen übertragen. Wir zeigen das am Beispiel des Additionsalgorithmus.

Die Grundidee des Additionsalgorithmus formuliert das Schema

$$\begin{array}{cccc} & a_n & \cdots & a_0 \\ & b_n & \cdots & b_0 \\ + & c_n & \cdots & c_0 \\ \hline c_{n+1} & d_n & \cdots & d_0 \end{array}$$

Dabei sind $a_n \cdots a_0$ und $b_n \cdots b_0$ die Ziffern der beiden zu addierenden Zahlen, $c_{n+1}\, c_n \cdots c_0$ die Überträge (englisch „carries“) ($c_0 = 0$), und $c_{n+1}\, d_n \cdots d_0$ die Ziffern des Ergebnisses. Wenn die Darstellungen der zu addierenden Zahlen unterschiedliche Länge haben, fügt man zu der kürzeren Darstellung führende Nullen hinzu.

Proposition 10.2.2 *Seien $B, n \in \mathbb{N}$ mit $B \geq 2$. Weiter seien die folgenden Definitionen gegeben:*

$$\mathit{Ziffer} = \{0, \ldots, B-1\}$$

$$\delta \in \mathit{Ziffer} \times \mathit{Ziffer} \times \{0,1\} \to \mathit{Ziffer}$$
$$\delta(z_1, z_2, c) = \text{if } z_1 + z_2 + c < B \text{ then } z_1 + z_2 + c \text{ else } z_1 + z_2 + c - B$$

$$\gamma \in \mathit{Ziffer} \times \mathit{Ziffer} \times \{0,1\} \to \{0,1\}$$
$$\gamma(z_1, z_2, c) = \text{if } z_1 + z_2 + c < B \text{ then } 0 \text{ else } 1$$

Schließlich seien $a_0, \ldots, a_n \in \mathit{Ziffer}$ und $b_0, \ldots, b_n \in \mathit{Ziffer}$ gegeben und $c_1, \ldots, c_{n+1} \in \{0,1\}$ wie folgt definiert:

$$c_0 = 0$$
$$c_i = \gamma(a_{i-1}, b_{i-1}, c_{i-1})$$

Dann gilt:

$$\sum_{i=0}^{n} a_i \cdot B^i + \sum_{i=0}^{n} b_i \cdot B^i = c_{n+1} \cdot B^{n+1} + \sum_{i=0}^{n} \delta(a_i, b_i, c_i) \cdot B^i$$

10.2.3 Realisierung von großen Zahlen

Wir werden große natürliche Zahlen als Listen von Ziffern repräsentieren:

```
type bignat = int list
```

Für die Algorithmen ist es zweckmäßig, eine Zahldarstellung

$$a_n \cdots a_0$$

in umgekehrter Reihenfolge

$$a_0 \cdots a_n$$

als Liste zu repräsentieren, damit die Rekursionen mit der niedrigwertigsten Stelle beginnen. Weiter legen wir fest, dass die Listenrepräsentation $a_n \neq 0$ erfüllt, also keine „führenden“ Nullen enthält. Diese Repräsentation hat drei Eigenschaften:

1. Die leere Liste repräsentiert die Null.
2. Die Listenrepräsentationen zweier großer Zahlen sind gleich genau dann, wenn sie dieselbe Zahl darstellen.
3. Wir müssen Listenrepräsentationen unterschiedlicher Länge addieren.

Aus Effizienzgründen arbeiten wir mit einer möglichst großen Basis. Dabei stellen wir sicher, dass das Produkt von zwei Ziffern noch eine kleine Zahl ist (also ein Wert von `int`). Mit Moscow ML auf PCs ist die größte mögliche Basis

$$\left\lfloor \sqrt{2^{30} - 1} \right\rfloor = 32767$$

Als Basis wählen wir

```
val B = 10000
```

da für Basen $B = 10^k$ die Konversion einer Darstellung zur Basis B in die Dezimaldarstellung besonders effizient realisierbar ist.[2] Für die Basis 10000 stellt die Liste

[2] Im Allgemeinen benötigt man für die Konversion in Dezimaldarstellung Division. Der Schulalgorithmus für Division ist für große Zahlen sehr ineffizient.

```
fun add'([],    br,    0)  = br
  | add'([],    br,    c)  = add'([c],br,0)
  | add'(ar,    [],    0)  = ar
  | add'(ar,    [],    c)  = add'(ar,[c],0)
  | add'(a::ar, b::br, c)  =
      let
        val s      = a+b+c
        val (d,c') = if s < B then (s, 0) else (s-B, 1)
      in
        d :: add'(ar,br,c')
      end

fun add(ar,br) = add'(ar,br,0)
val add : bignat * bignat -> bignat
```

Abbildung 10.4: Eine Prozedur zum Addieren großer natürlicher Zahlen

```
[2345, 456, 4]
```

die Zahl

```
404562345
```

dar. Diese Zahl ist bereits nicht mehr im Bereich von `int`.

Eine Prozedur, die kleine natürliche Zahlen in die Darstellung für große Zahlen konvertiert, können wir wie folgt schreiben:

```
fun fromInt n = if n<1 then []
                else n mod B :: fromInt(n div B)
val fromInt : int -> bignat

fromInt 394567834
[7834, 9456, 3] : bignat
```

Abbildung 10.4 zeigt eine Prozedur `add`, die große Zahlen addiert. Die eigentliche Arbeit erledigt dabei die Hilfsprozedur `add'`, die den aktuellen Übertrag als drittes Argument übergeben bekommt.

Um die Termination von `add'` zu zeigen, benötigen wir eine wohlfundierte Relation

$$\succ \subseteq \mathcal{L}(\mathbb{N}) \times \mathcal{L}(\mathbb{N}) \times \{0,1\}$$

Man überzeugt sich leicht, dass die Relation

$$(a,b,c) \succ (a',b',c') \iff \begin{pmatrix} |a|+|b|+c > |a'|+|b'|+c' \\ \vee \\ c > c' \ \wedge \ |a|+|b|+c = |a'|+|b'|+c' \end{pmatrix}$$

```
eqtype bignat

val fromString : string -> bignat
val toString   : bignat -> string

val add        : bignat * bignat -> bignat
val sub        : bignat * bignat -> bignat  (* Domain *)
val less       : bignat * bignat -> bool
val mul        : bignat * bignat -> bignat
val divmod     : bignat * bignat -> bignat * bignat  (* Div *)
```

Abbildung 10.5: Eine Signatur für große natürliche Zahlen

wohlfundiert ist und die Terminationsbedingungen für `add'` erfüllt.

Mit den gezeigten Ideen kann man eine Typabstraktion für große natürliche Zahlen mit der in Abbildung 10.5 gezeigten Signatur implementieren (siehe Aufgabe 10.4). Die Subtraktionsoperation `sub` soll die vordefinierte Ausnahme `Domain` werfen, wenn das zweites Argument größer ist als das erste. Die Divisionsoperation `divmod` soll zu $(x, y) \in \mathbb{N} \times \mathbb{N}^+$ das Paar $(\lfloor x/y \rfloor\,,\ x \bmod y)$ liefern. Für $(x, 0)$ soll `divmod` die Ausnahme `Div` werfen.

Große ganze Zahlen können wir als Paare aus einem Vorzeichen (plus oder minus) und einer großen natürlichen Zahl darstellen. Die Operationen für große ganze Zahlen können durch Rückgriff auf die Operationen für große natürliche Zahlen geschrieben werden.

Große rationale Zahlen können wir als Paare (Zähler, Nenner) aus zwei großen ganzen Zahlen darstellen. Dabei ist es sinnvoll, eine gekürzte Darstellung zu wählen.

10.3 Das Damenproblem

Das Damenproblem hat schon den Mathematiker Gauß beschäftigt.[3] Die Aufgabe besteht darin, auf einem Schachbrett 8 Damen so aufzustellen, dass sie sich nicht gegenseitig schlagen können.[4] Abbildung 10.6 zeigt eine Lösung des 8-Damenproblems. Allgemeiner interessiert uns das *n-Damenproblem*, bei dem n Damen auf einem $n \times n$ Schachbrett so aufgestellt werden sollen, dass sie sich nicht gegenseitig schlagen können. Das 1-Damenproblem ist trivial lösbar. Man überzeugt sich leicht, dass das Damenproblem für $n = 2$ und $n = 3$ unlösbar

[3] Carl Friedrich Gauß, 1777–1855.

[4] Zwei Damen können sich schlagen, wenn sie in derselben Zeile, Spalte oder Diagonale stehen.

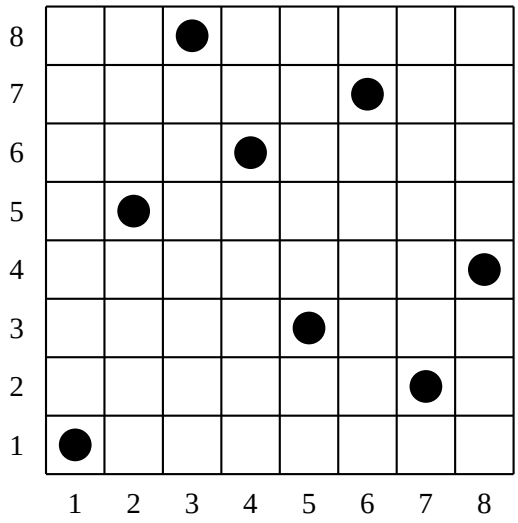


Abbildung 10.6: Eine Lösung des 8-Damenproblems

ist. Schlaue Leute haben bewiesen, dass das n-Damenproblem für $n \geq 4$ immer lösbar ist.[5]

Repräsentation der Lösungen

Zuerst überlegen wir uns eine Repräsentation für die Lösungen des Damenproblems. Eine Lösung kann durch die Koordinaten der gesetzten Damen repräsentiert werden. Da jede Lösung in jede Spalte genau eine Dame setzen muss, können wir die Koordinatenpaare einer Lösung nach den Spalten ordnen

$$(1, y_1)\ ,\ \ldots\ ,\ (n, y_n)$$

und kompakt durch die Liste

$$[y_1, \ldots, y_n]$$

der y-Koordinaten darstellen. Zum Beispiel kann die Lösung in Abbildung 10.6 durch die Liste

[1, 5, 8, 6, 3, 7, 2, 4]

dargestellt werden.

Formale Definition des Problems

Als nächstes schreiben wir eine Prozedur

[5] Bernd-Jürgen Falkowski und Lothar Schmitz, A Note on the Queens Problem. Information Processing Letters 23, 1986, 39–46.

```
fun solution n xs = permutation n xs andalso safe xs
val solution : int -> int list -> bool
```

die testet, ob eine Liste eine Lösung des n-Damenproblems ist. Die Bedingung `permutation n xs` testet, dass `xs` eine Permutation der Liste $[1, \ldots, n]$ ist. Damit ist sichergestellt, dass n Damen innerhalb des Bretts so platziert sind, dass in jeder Spalte und in jeder Zeile genau eine Dame steht. Die zweite Bedingung `safe xs` stellt sicher, dass sich die Damen nicht diagonal schlagen können. Hier sind die entsprechenden Prozeduren:

```
fun list1to n = List.tabulate(n, fn i=>i+1)
val list1to : int -> int list

fun permutation n xs =
    Listsort.sort Int.compare xs = list1to n
val permutation : int -> int list -> bool

fun safe'(_, _, nil  ) = true
  | safe'(y, i, x::xr) = y-i<>x andalso y+i<>x
                         andalso safe'(y, i+1, xr)
val safe' : int list * int * int -> bool

fun safe nil     = true
  | safe (x::xr) = safe'(x,1,xr) andalso safe xr
val safe : int list -> bool
```

Lösungsstrategie

Um das n-Damenproblem zu lösen, gehen wir wie folgt vor: Zuerst setzen wir eine Dame in die n-te Spalte, dann in die $n-1$-te, und so weiter. Wenn wir Glück haben, bekommen wir so eine Lösung des Problems. Wenn wir kein Glück haben, erreichen wir einen Zustand, bei dem keine weitere Dame platziert werden kann. Dann müssen wir *zurücksetzen* und eine der bereits platzierten Damen umsetzen.[6] Wir lösen das Problem also durch den schrittweisen Aufbau von *Teillösungen* und durch systematisches Probieren. Im Fachjargon sagt man, dass das Problem durch *Suche* gelöst wird.

Zustände

Mit der Prozedur

```
fun state n (xs,ys,zs) =
    permutation n (xs@ys@zs) andalso safe zs
val state : int -> int list * int list * int list -> bool
```

definieren wir, was wir unter einem *Zustand* verstehen wollen. Ein Zustand

```
(xs,ys,zs)
```

[6] Im Englischen spricht man von „backtracking“.

partitioniert die Zeilennummern $1, \ldots, n$ (also die y-Koordinaten der zu setzenden Damen) in drei Listen. Die Liste `zs` repräsentiert die schon gesetzten Damen (die Teillösung des Zustands) und die Liste `ys` enthält die Zeilennummern, die für die nächste zu setzende Dame noch in Frage kommen. (Erinnerung: wir setzen die Damen von rechts nach links.)

Der *Anfangszustand* ist

```
(nil, list1to n, nil)
```

Es gibt zwei Arten von Endzuständen. *Erfolgreiche Endzustände* haben die Form

```
(nil, nil, zs)
```

und die schöne Eigenschaft, dass `zs` eine Lösung des n-Damenproblems ist (das folgt aus unserer Definition von Zuständen mit `state`). *Gescheiterte Endzustände* haben die Form

```
(x::xs, nil, zs)
```

Wenn wir einen gescheiterten Zustand erreichen, müssen wir zu einem der vorhergegangenen Zustände zurücksetzen.

Transitionsregeln

Wir benötigen zwei *Transitionsregeln*, die die Übergänge von einem Zustand in seine möglichen *Folgezustände* festlegen:

1. Von einem Zustand `(xs,y::yr,zs)` kann man zu dem Zustand `(nil,xs@yr,y::zs)` übergehen, genau dann, wenn die Gleichung `safe'(y,1,zs)=true` gilt.
2. Von einem Zustand `(xs,y::yr,zs)` kann man immer zu dem Zustand `(y::xs,yr,zs)` übergehen.

Für einen Zustand, der kein Endzustand ist, gibt es wenigstens einen Folgezustand, und für Endzustände gibt es keine Folgezustände. Außerdem sind die Transitionsregeln so beschaffen, dass spätestens nach n^2 Schritten ein Endzustand erreicht wird (gescheitert oder erfolgreich).[7] Wir sagen, dass die Transitionsregeln *terminierend* sind. Schließlich können wir ausgehend vom Anfangszustand

```
(nil, list1to n, nil)
```

zu jeder vorgegebenen Lösung durch entsprechendes *Anwenden der Transitionregeln* einen erfolgreichen Endzustand erreichen, der diese Lösung enthält. Wir sagen, dass die Transitionsregeln *vollständig* sind. Die Definition von Zuständen

[7] Die Zustände werden bei einem Übergang mit den Transitionsregeln in einem abstrakten Sinne kleiner, da entweder die letzte Komponente verlängert (diese wird nie verkürzt) oder die mittlere Komponente verkürzt wird (diese wird nur verlängert, wenn gleichzeitig die letzte Komponente verlängert wird).

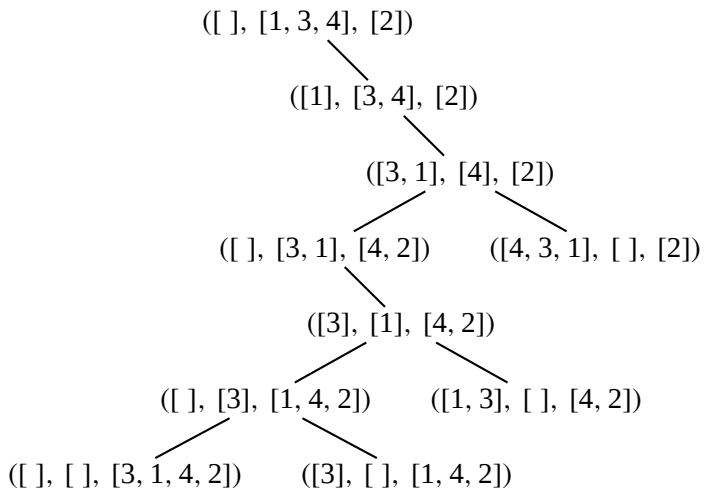


Abbildung 10.7: Der Transitionsbaum für den Zustand ([ ], [1, 3, 4], [2])

und Transitionsregeln bezeichnen wir zusammenfassend als ein *Transitionssystem*.

Transitionsbäume

Abbildung 10.7 zeigt alle Zustände, die mit Hilfe der Transitionsregeln vom Zustand ([ ], [1, 3, 4], [2]) aus erreichbar sind. Gemäß ihrer Erreichbarkeit durch die Transitionsregel lassen sich diese Zustände zu einem sogenannten *Transitionsbaum* anordnen, der die Form eines S-Baums hat. Zustände, die mit der ersten Regel erreicht werden, sind nach links gezeichnet, und Zustände, die mit der zweiten Regel erreicht werden, sind nach rechts gezeichnet Der Ausgangszustand erscheint als Wurzel und die Endzustände als Blätter. Der linkeste Endzustand ist erfolgreich, die anderen drei sind gescheitert. Der erfolgreiche Endzustand enthält eine Lösung des Damenproblems für $n = 4$.

Berechnung aller Lösungen

Mit diesen Überlegungen ist es einfach, die in Abbildung 10.8 gezeigte Prozedur zu schreiben, die alle Lösungen des n-Damenproblems liefert.

```
solve 0
val [[]] : int list list

solve 1
val [[1]] : int list list

solve 2
val [] : int list list

solve 3
val [] : int list list
```

```
fun solve n =
    let
        fun solve' (nil  , nil  , zs) = [zs]
          | solve' (x::xr, nil  , _ ) = nil
          | solve' (xs   , y::yr, zs) =
                    (if safe'(y,1,zs)
                     then solve'(nil, xs@yr, y::zs)
                     else nil) @ solve'(y::xs, yr, zs)
    in
        solve'(nil, list1to n, nil)
    end
val solve : int -> int list list
```

Abbildung 10.8: Berechnung aller Lösungen des n-Damenproblems

```
solve 4
val [[3, 1, 4, 2], [2, 4, 1, 3]] : int list list

length(solve 8)
val 92 : int

hd(solve 8)
val [4, 2, 7, 3, 6, 8, 5, 1] : int

length(solve 11)
val 2680 : int
```

Berechnung einer Lösung

Für größer werdende n nimmt die Anzahl der Lösungen sehr schnell zu und unser Löser `solve` rechnet sehr lange. Um auch für größere n noch eine Lösung in erträglicher Zeit zu bekommen, schreiben wir `solve` so um, dass `solve'` die Suche nach dem Finden der ersten Lösung durch das Werfen einer Ausnahme, die die Lösung enthält, abbricht. Da `solve'` dann nur noch die leere Liste als mögliches Ergebnis hat, ersetzen wir diese durch das leere Tupel `()` (das einzige Element des Typs `unit`). Damit bekommen wir die in Abbildung 10.9 gezeigte Prozedur.

```
first 2
NONE : int list option

first 16
SOME [11, 9, 7, 4, 8, 14, 16, 6, 15, 12, 10, 13, 2, 5, 3, 1]
 : int list option
```

Für noch größere n rechnet aber auch `first` sehr lange. Probieren Sie 24 und 25. In dem in Fußnote 5 erwähnten Artikel ist ein Algorithmus angegeben, der das Problem auch für große n schnell löst. Dieser Algorithmus kann mit etwa 50 Zeilen in Standard ML geschrieben werden.

```
fun first n =
    let
        fun safe'(_, _, nil  ) = true
          | safe'(y, i, x::xr) = y-i<>x andalso y+i<>x
                                 andalso safe'(y, i+1, xr)

        exception Solved of int list

        fun first' (nil  , nil  , zs) = raise Solved zs
          | first' (x::xr, nil  , _ ) = ()
          | first' (xs   , y::yr, zs) =
            ( if safe'(y,1,zs) then first'(nil, xs@yr, y::zs)
              else () ;
              first'(y::xs, yr, zs) )
    in
        (first'(nil, List.tabulate(n, fn i=>i+1), nil); NONE)
        handle Solved xs => SOME xs
    end
val first : int -> int list option
```

Abbildung 10.9: Berechnung einer Lösung des n-Damenproblems

Übungen

Aufgabe 10.1 (Rot-Schwarz-Bäume)

1. Schreiben Sie eine Prozedur

   ```
   listA : ctree -> int list
   ```

   die zu einem Rot-Schwarz-Baum eine aufsteigend sortierte Liste liefert, die die Elemente der dargestellten Menge enthält. Die Laufzeit der Prozedur soll linear bezüglich der Größe der dargestellten Mengen sein.

2. Schreiben Sie eine Prozedur

   ```
   listD : ctree -> int list
   ```

   die zu einem Rot-Schwarz-Baum eine absteigend sortierte Liste liefert, die die Elemente der dargestellten Menge enthält. Die Laufzeit der Prozedur soll linear bezüglich der Größe der dargestellten Mengen sein.

3. Schreiben Sie eine Prozedur

   ```
   sb : ctree -> bool
   ```

   die testet, ob ein C-Baum (als S-Baum betrachtet) ein Suchbaum ist.

4. Schreiben Sie eine Prozedur

```
rb : ctree -> int (* NotRB *)
```

die die S-Höhe eines RB-Baums liefert. Wenn das Argument kein RB-Baum ist, soll die Ausnahme `NotRB` geworfen werden. Gehen Sie dabei ähnlich vor wie beim schnellen Test auf Balanciertheit (siehe Abschnitt 6.8). Verwenden Sie außerdem eine Hilfsprozedur

```
black : ctree -> unit (* NotRB *)
```

die für einen C-Baum die Ausnahme `NotRB` wirft, wenn seine Wurzel rot ist.

Aufgabe 10.2 (Funktor für effiziente endliche Mengen) Schreiben Sie analog zu Abschnitt 9.4 einen Funktor `Set`, der endlicher Mengen mit Rot-Schwarz-Bäumen implementiert.

Aufgabe 10.3 (Funktor für effiziente endliche Funktionen) Schreiben Sie analog zu Aufgabe 9.2 einen Funktor `Map`, der endliche Funktionen mit Rot-Schwarz-Bäumen implementiert. Verwenden Sie dabei einen Konstruktortyp für C-Bäume, der Ihnen erlaubt, die in Abbildung 10.3 gezeigten Rotationsprozeduren ohne Änderung zu übernehmen. Verwenden Sie zudem eine Hilfprozedur `compare'`, die Ihnen erlaubt, die Prozedur `insert'` in Abbildung 10.3 zu übernehmen, nachdem Sie `Int.compare` durch `compare'` ersetzt haben.

Aufgabe 10.4 (Große natürliche Zahlen) Implementieren Sie eine Typabstraktion für große natürliche Zahlen gemäß der Signatur in Abbildung 10.5. Gehen Sie dabei so wie in Abschnitt 10.2 und nachfolgend beschrieben vor.

Im Folgenden benutzen wir das Typsynonym

```
type digit = int
```

für die Menge der Ziffern bezüglich der Basis B (also die Zahlen 0 bis $B-1$).

Da `bignat` als Typ mit Gleichheit sichtbar ist, dürfen keine führenden Nullen eingeführt werden. Dabei ist eine Hilfsprozedur

```
adj : digit * bignat -> bignat
```

nützlich, die zu (d, x) die Zahl $d + x\,B$ berechnet.

Verwenden Sie eine Hilfsprozedur

```
fromInt : int -> bignat
```

Implementieren Sie die Operation `mul` mit einer Hilfsprozedur

```
mul' : bignat * digit * digit -> bignat
```

die zu (x, b, c) die Zahl $xb + c$ berechnet. Die Prozedur `mul` soll gemäß der Gleichung

$$x(y + y'B) = xy + (xy')B$$

realisiert werden.

Implementieren Sie die Operation `divmod` mit einer Hilfsprozedur

```
divmod' : bignat * bignat * digit -> digit * bignat
```

die zu $(x, y, 0)$ mit $x < yB$ das Paar $(\lfloor x/y \rfloor, x \bmod y)$ berechnet. Die Prozedur `divmod` soll gemäß den Gleichungen

$$\left\lfloor \frac{x}{y} \right\rfloor = \left\lfloor \frac{x}{yB} \right\rfloor B + \left\lfloor \frac{x \bmod (yB)}{y} \right\rfloor$$

$$x \bmod y = (x \bmod (yB)) \bmod y$$

realisiert werden. Beachten Sie, dass $x \bmod (yB) < yB$ gilt.

Leider ist der Schulagorithmus für Division für größere Zahlen ineffizient. Implementieren Sie daher `toString` ohne Division indem Sie ausnützen, dass Sie mit der Basis 10000 arbeiten. Alle anderen Operationen sollen jedoch so realisiert werden, dass sie für beliebige Basen arbeiten.

Realiseren Sie Ihre Implementierung großer natürlicher Zahlen durch eine Komponente `Bignat`. Schreiben Sie mithilfe von `Bignat` ein direkt ausführbares Programm, dass zu einem Startargument $n \in \mathbb{N}$ die Fakultät $n!$ ausgibt. Berechnen Sie damit 1000!.