Kapitel 7

Korrektheit und Laufzeit

Sei eine Funktion f gegeben. Dann kann man nach einer Prozedur p fragen, die f berechnet. Gegeben diese Situation, nennt man die Funktion f eine *Spezifikation* und die Prozedur p eine *Implementierung* der Spezifikation.

Gegeben eine Spezifikation f und eine Implementierung p, stellt man zwei Fragen:

1. *Korrektheit*: Berechnet die Prozedur p tatsächlich die Funktion f?
2. *Laufzeit*: Wieviel Zeit benötigt die Prozedur p für die Berechnung der Funktion f?

Offensichtlich sind wir nur an korrekten Implementierungen interessiert. Korrektheit alleine genügt aber nicht. Wenn die Berechnung der Prozedur zuviel Zeit braucht (zum Beispiel 1000 Jahre), ist sie praktisch wertlos.

Für die Außenansicht von Prozeduren ist das *Blackbox-Modell* hilfreich. Dieses stellt eine Prozedur als ein Kästchen mit einer Eingabe- und einer Ausgabeöffnung dar. Wenn wir in die Eingabeöffnung einen Wert stecken, beginnt das Kästchen zu rechnen. Es gibt dann zwei Möglichkeiten: Entweder rechnet das Kästchen unendlich lange (Divergenz), oder es legt nach einer gewissen Zeit einen Wert in die Ausgabeöffnung (Terminierung). In diesem Kapitel werden wir nur Prozeduren betrachten, die keine Ausnahmen werfen.

Unter einem *Prozeduraufruf* verstehen wir ein Paar aus einer Prozedur und einem Eingabewert.[1] Die *Laufzeit eines Prozeduraufrufs* ist die Zeit, die bis zur Lieferung des Ausgabewerts benötigt wird. Wir nehmen an, dass ein Prozeduraufruf

[1] Prozeduraufrufe unterscheiden sich von Prozeduranwendungen. Eine Prozeduranwendung ist ein Ausdruck $a_1\ a_2$, der aus zwei Unterausdrücken a_1 und a_2 besteht. Wenn man beide Unterausdrücke auswertet, ergibt das Paar der beiden Werte einen Prozeduraufruf.

immer dieselbe Laufzeit benötigt. Die *Laufzeit einer Prozedur* ist die Funktion, die die Eingabewerte der Prozedur auf ihre jeweilige Laufzeit abbildet.

In diesem Kapitel lernen wir Techniken, mit denen wir prinzipielle Aussagen über die Laufzeit von Prozeduren machen können. Für diese Aussagen spielt die Geschwindigkeit des ausführenden Computers keine Rolle. Auch die Details der Programmierssprache, in der die Prozedur formuliert ist, sind irrelevant.

Prozeduren sind passive Objekte, mit denen ein Interpreter rechnet. Weil es aber bequem und allgemein üblich ist, werden wir im Folgenden oft *operationale Sprechweisen* für Prozeduren verwenden. Zum Beispiel sagen wir, dass eine Prozedur schneller rechnet als eine andere oder dass eine Prozedur eine andere Prozedur aufruft.

Teilsprache FML

Im diesem Kapitel betrachten wir eine Teilsprache FML der bisher eingeführten Teilsprache von Standard ML wie folgt:

1. Keine Gleitkommazahlen.
2. Keine Ausnahmen.

Die dynamische Semantik von FML ist mithilfe von Auswertungsprotokollen definiert (siehe Abschnitte 1.3 und 3.9). Dabei werden keine Größenbeschränkungen für ganze Zahlen und keine Speicherbeschränkungen auferlegt. Diese Beschränkungen werden durch den Interpreter eingeführt. Wir unterscheiden also zwischen Ausführung gemäß der Sprache FML und Ausführung gemäß eines Interpreters. Wenn wir keinen Interpreter erwähnen, meinen wir mit Ausführung stets Ausführung gemäß der Sprache.

7.1 Prozeduren und Funktionen

Eine wichtige Eigenschaft von FML ist die Tatsache, dass wir jede monomorphe Prozedurdeklaration auch als eine Funktionsdefinition ansehen können. Hier ist ein Beispiel:

```
fun sum(x,y) = if x=0 then y else 1+sum(x-1,y)
val sum : int * int -> int
```

$sum \in \mathbb{Z} \times \mathbb{Z} \rightharpoonup \mathbb{Z}$

$sum(x, y) = \text{if } x = 0 \text{ then } y \text{ else } 1 + sum(x - 1, y)$

Zwischen der Prozedur `sum` und der Funktion *sum* besteht ein sehr enger Zusammenhang:

> Für $(x, y) \in \mathbb{Z}^2$ terminiert die Prozedur `sum` genau dann mit einer Zahl $z \in \mathbb{Z}$, wenn $sum(x, y) = z$ gilt.

Die Prozedur `sum` ist eine algorithmische Beschreibung der Funktion *sum*. Die Funktion *sum* ist lediglich eine Teilmenge von $(\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z}$. Es gibt viele Möglichkeiten die Funktion *sum* zu beschreiben. Eine davon ist:

$$sum \;=\; \lambda\,(x, y) \in \mathbb{N} \times \mathbb{Z}\,.\; x + y$$

Beweis Offensichtlich divergiert die Prozedur `sum` für $(x, y) \in \mathbb{Z}^2$ mit $x \notin \mathbb{N}$. Also gilt $Dom\ sum \subseteq \mathbb{N} \times \mathbb{Z}$. Sei $y \in \mathbb{Z}$. Wir zeigen durch Induktion über x:

$$\forall x \in \mathbb{N}\colon\ sum(x, y) = x + y$$

Sei $x = 0$. Aus der Definition von *sum* folgt: $sum(x, y) = sum(0, y) = y = 0 + y = x + y$.

Sei $x > 0$. Dann

$$\begin{aligned} sum(x, y) &= 1 + sum(x - 1, y) && \text{Definition } sum \\ &= 1 + (x - 1) + y && \text{Induktionsannahme} \\ &= x + y \end{aligned}$$

□

Wir wissen jetzt, dass die Prozedur `sum` für alle $(x, y) \in \mathbb{N} \times \mathbb{Z}$ terminiert und die Zahl $x + y$ als Ergebnis liefert. Die Prozedur `sum` ist also eine unnötig komplizierte Berechnungsvorschrift für die Funktion $\lambda\,(x, y) \in \mathbb{N} \times \mathbb{Z}\,.\; x + y$.

Wenn wir die Prozedur `sum` mit einem Interpreter ausführen, der Speicherbeschränkungen und Größenbeschränkungen für ganze Zahlen auferlegt, wird die obige Aussage nur teilweise invalidiert. Für diesen Fall wissen wir immerhin noch die folgenden Dinge:

1. Die Prozedur `sum` terminiert für alle $(x, y) \in \mathbb{N} \times \mathbb{Z}$.
2. Wenn die Prozedur `sum` für ein Paar $(x, y) \in \mathbb{N} \times \mathbb{Z}$ regulär terminiert, dann liefert sie den Wert $x + y$.

Betrachten Sie noch einmal die Definition der Funktion *sum*

$$sum \in \mathbb{Z} \times \mathbb{Z} \rightharpoonup \mathbb{Z}$$
$$sum(x, y) = \text{if } x = 0 \text{ then } y \text{ else } 1 + sum(x - 1, y)$$

Offensichtlich können wir diese Funktionsdefinition auch als eine Prozedurdefinition auffassen. Aus mathematischer Sicht ist es unerheblich, ob eine Prozedur in einer konkreten Programmiersprache oder mithilfe von mathematischer Notation definiert ist. In der Tat haben Logiker rekursive Prozeduren schon in der ersten Hälfte des zwanzigsten Jahrhunderts untersucht, lange bevor es Programmiersprachen und Computer gab.

Wir können also eine Funktionsdefinition auf zwei Arten interpretieren: *funktional* und *prozedural*. Die prozedurale Interpretation liefert eine Prozedur, die gemäß der Definition beschreibt, wie die Funktion zu berechnen ist. Die funktionale Interpretation liefert eine Funktion, die offen lässt, wie sie berechnet werden soll.

Die meisten Programmiersprachen haben die Eigenschaft, dass man in ihnen bestimmte Funktionsdefinitionen als Prozedurdeklarationen formulieren kann. Die so definierten Prozeduren berechnen dann die entsprechenden Funktionen und werden als *funktionale Prozeduren* bezeichnet. In Standard ML kann man, verglichen mit anderen Programmiersprachen, eine besonders große Klasse von Funktionsdefinitionen als Prozedurdeklarationen formulieren.

Wenn wir eine Funktionsdefinition in einer Programmiersprache als Prozedurdeklaration formulieren, sind wir mithilfe eines Computers in der Lage, die Funktion zu berechnen. Zudem kann automatisch überprüft werden, ob die Prozedurdeklaration nach den Regeln der Programmiersprache zulässig ist. Ein Standard ML Interpreter ist außerdem in der Lage, den Typ der Prozedur zu bestimmen.

Funktionale Prozeduren haben den Vorteil, dass viele ihrer Eigenschaften *unabhängig* von der Programmiersprache bewiesen werden können. Wir haben dies am Beispiel der Prozedur `sum` gesehen.

Die Teilsprache FML hat, wie bereits erwähnt, die Eigenschaft, dass alle monomorphen Prozedurdeklarationen als Funktionsdefinitionen aufgefasst werden können und entsprechende funktionale Prozeduren definieren. Programmiersprachen mit dieser Eigenschaft nennt man *streng funktional*. Wir werden später Sprachkonstrukte kennenlernen, mit denen nicht-funktionale Prozeduren definiert werden können.

Für FML definieren wie die folgenden Begriffe.

Definition 7.1.1 *Sei die Prozedur p durch eine monomorphe Prozedurdeklaration definiert. Dann bezeichnen wir die Funktion, die durch die der Prozedurdeklaration entsprechende Funktionsdefinition definiert wird, mit f_p und nennen sie* die der Prozedur p zugeordnete Funktion. *Wir sagen, dass die Prozedur p eine Funktion f* berechnet *genau dann, wenn $f \subseteq f_p$.*

Ein Typ heißt *elementar*, wenn er ohne den Typkonstruktor `->` gebildet ist. Hier sind Beispiele für elementare Typen:

```
int
int * bool
int list
((int list) * bool) list
```

Eine monomorphe Prozedur des Typs t_1 `->` t_2 heißt *elementar*, wenn t_1 und t_2 elementare Typen sind.

Proposition 7.1.2 *Für jede elementare Prozedur* p: t_1 `->` t_2 *gilt:*

1. *Dom* f_p *ist die Menge aller Werte des Typs* t_1*, für die* p *terminiert.*
2. $f_p(w) = w'$ *gilt genau dann, wenn* p *für* w *mit dem Ergebnis* w' *terminiert.*

Die einer elementaren Prozedur zugeordnete Funktion beschreibt also das Ein-Ausgabe-Verhalten der Prozedur.

Für nicht-elementare Prozeduren besteht ebenfalls ein enger Zusammenhang zwischen Prozedur und zugeordneter Funktion, der aber etwas aufwendiger zu formulieren ist. Wir werden darauf später zurückkommen.

Vorsicht: Partielle Funktionen

In diesem Kapitel arbeiten wir oft mit Ausdrücken und Gleichungen, die möglicherweise undefinierte Anwendungen von partiellen Funktionen enthalten. Undefinierte Funktionsanwendungen kann man analog zu divergierenden Prozeduranwendungen verstehen. Mit dem Symbol $\perp$ (sprich „Div“) bezeichnen wir undefinierte Ausdrücke. Die folgenden Beispiele zeigen, wie man mit $\perp$ rechnen kann (*sum* ist die oben definierte Funktion $\mathbb{Z}^2 \rightharpoonup \mathbb{Z}$):

$sum(0, 5) = 5 \neq \perp$

$sum(-1, 5) = \perp \neq 5$

$\perp + 13 = \perp$

$0 \cdot sum(-1, 5) = 0 \cdot \perp = \perp \neq 0$

$sum(2, sum(-1, 5)) = sum(2, \perp) = \perp$

if $1 < 2$ then 1 else $\perp = 1$

if $\perp$ then 1 else $2 = \perp$

Fakultätsfunktion

Rekursive Funktionsdefinitionen werden oft zusammen mit einer Zusicherung formuliert. Als Beispiel betrachten wir die folgende Definition der *Fakultätsfunktion*:

$$
\begin{aligned}
&! \in \mathbb{N} \to \mathbb{N} \\
&0! = 1 \\
&n! = n \cdot (n-1)! \quad \text{falls } n > 0
\end{aligned}
$$

Diese besteht aus der *eigentliche Definition*

$$
\begin{aligned}
&! \in \mathbb{N} \rightharpoonup \mathbb{N} \\
&0! = 1 \\
&n! = n \cdot (n-1)! \quad \text{falls } n > 0
\end{aligned}
$$

und der *Zusicherung*

$$\forall n \in \mathbb{N}\colon\ n! \in \mathbb{N}$$

Die eigentliche Definition entspricht bis auf die unterschiedliche Typisierung der Prozedurdeklaration

```
fun fak n = if n=0 then 1 else n*fak(n-1)
val fak : int -> int
```

Die Zusicherung ist eine Behauptung, die bewiesen werden muss. Sie sagt, dass die durch die eigentliche Definition festgelegte Funktion ! für alle $n \in \mathbb{N}$ definiert sein muss. Unsere Intuition für Prozeduren sagt uns, dass dies der Fall ist, da die durch die eigentliche Definition definierte Prozedur ! für alle $n \in \mathbb{N}$ terminiert. Man kann die Zusicherung aber auch durch Induktion über n beweisen.

Fibonacci-Zahlen

Die sogenannten *Fibonacci-Zahlen*,[2] sind wie folgt definiert:

$$
\begin{aligned}
&f_0 = 0 \\
&f_1 = 1 \\
&f_n = f_{n-1} + f_{n-2} \quad \text{falls } n > 1
\end{aligned}
$$

Offensichtlich wird hier eine Funktion $f \in \mathbb{N} \to \mathbb{N}$ rekursiv definiert und die Funktionsanwendung $f(i)$ als f_i geschrieben. Die entsprechende Prozedurdeklaration sieht in Standard ML wie folgt aus:

[2] Die Fibonacci-Zahlen spielen eine wichtige Rolle in der Zahlentheorie. Wenn Sie mehr darüber wissen wollen, empfehlen wir Ihnen das Buch von Graham, Knuth und Patashnik [**?**].

```
fun fib n = if n=0 then 0
            else if n=1 then 1
                 else fib(n-1) + fib(n-2)
val fib : int -> int
```

Alternativ können wir auch die Prozedur

```
fun fib' n = if n<2 then n else fib'(n-1) + fib'(n-2)
val fib' : int -> int
```

schreiben, die für alle $n \in \mathbb{Z}$ terminiert und ebenfalls die Funktion $\lambda\, i \in \mathbb{N}.\ f_i$ berechnet.

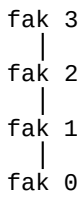
7.2 Rekursionsbäume

Für die Ausführung eines Aufrufs einer rekursiven Prozedur müssen in der Regel weitere Aufrufe dieser Prozedur ausgeführt werden. Die Gesamtheit dieser Aufrufe kann als ein geordneter Baum angeordnet werden, der als *Rekursionsbaum* bezeichnet wird. Der initiale Aufruf dient dabei als Wurzel. Die Nachfolger eines Aufrufs bilden die für die Ausführung dieses Aufrufs direkt benötigten rekursiven Aufrufe. Diese werden in der Reihenfolge ihrer Ausführung angeordnet, wobei der zuerst ausgeführte Nachfolger ganz links steht.

Wir betrachten einige Beispiele. Wir beginnen mit der Fakultätsprozedur

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

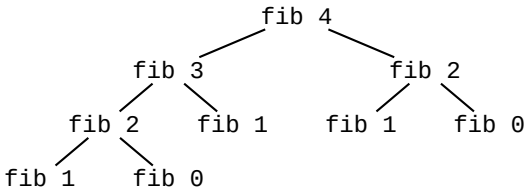
Für den Aufruf `fak` 3 erhalten wir den Rekursionsbaum



Als Nächstes betrachten wir die Fibonacci-Prozedur

```
fun fib n = if n<2 then n else fib(n-1) + fib(n-2)
```

Für den Aufruf `fib` 4 erhalten wir den Rekursionsbaum



Rekursionstiefe

Die Tiefe des Rekursionsbaums eines Prozeduraufrufs bezeichnet man als die *Rekursionstiefe des Prozeduraufrufs*.

lineare Rekursion

Eine Prozedur heißt *linear-rekursiv* genau dann, wenn alle ihre Rekursionsbäume lineare Bäume sind. Die Prozedur `fak` ist ein Beispiel für eine linear-rekursive Prozedur. Nicht-lineare Rekursion wird als *Baumrekursion* bezeichnet.

Baumrekursion

binäre Rekursion

Eine Prozedur heißt *binär-rekursiv* genau dann, wenn alle ihre Rekursionsbäume binäre Bäume sind. Die Prozedur `fib` ist ein Beispiel für eine binär-rekursive Prozedur.

Ackermann-Funktion

Die Prozedur

```
fun am(x,y) = if x=0 then y+1
              else if y=0 then am(x-1,1)
                   else am(x-1, am(x,y-1))
val am : int * int -> int
```

berechnet eine Variante der sogenannten *Ackermann-Funktion*.[3] Diese Prozedur ist baumrekursiv. Sie ist mithilfe einer *geschachtelten Rekursion*

geschachtelte Rekursion

```
am(x-1, am(x,y-1))
```

definiert. Bei einer geschachtelten Rekursion ist eine rekursive Anwendung in eine zweite rekursive Anwendung geschachtelt. Abbildung 7.1 zeigt den Rekursionsbaum für den Aufruf `am`$(2, 2)$. Wegen der geschachtelten Rekusion haben wir die Aufrufe zusammen mit ihren Ergebnissen angegeben.

7.3 Terminierungsbeweise

In Abschnitt 7.1 haben wir mithilfe von Induktion bewiesen, dass die Prozedur `sum` für bestimmte Argumente terminiert. Wir geben jetzt eine alternative Technik für Terminierungsbeweise an, die ohne Induktion auskommt.

Ein Aufruf einer rekursiven Prozedur terminiert offensichtlich dann, wenn die Argumente der rekursiven Aufrufe im Rekursionsbaum jeweils in einem gewissen

[3] Die Ackermann-Funktion spielt in der Theorie der berechenbaren Funktionen (Rekursionstheorie) eine wichtige Rolle. Sie ist nach dem Logiker Wilhelm Ackermann, 1896–1962, benannt.

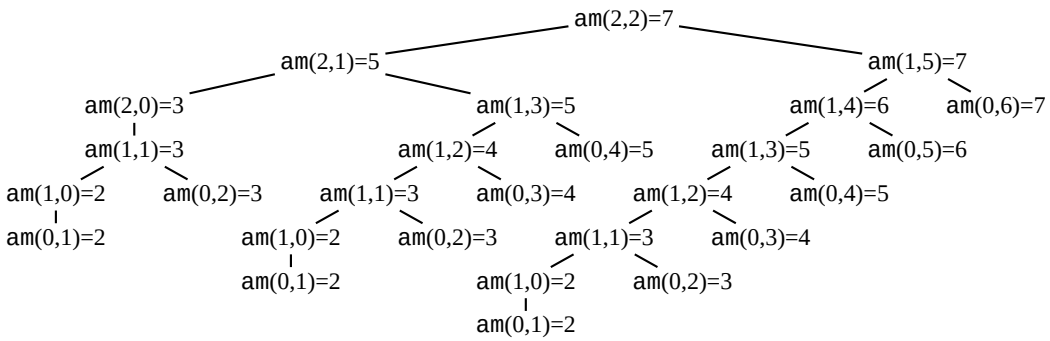


Abbildung 7.1: Ein Rekursionsbaum für die Ackermann-Prozedur

Sinne kleiner werden. Die Idee des Kleinerwerdens wird durch den Begriff der wohlfundierten Relation formalisiert.

wohlfundierte Relationen

Sei X eine Menge. Eine *wohlfundierte Relation auf X* ist eine Menge $\succ \subseteq X \times X$, die die Eigenschaft hat, dass keine unendliche Kette

$$x_0 \succ x_1 \succ x_2 \succ x_3 \succ x_4 \succ \ldots$$

existiert. Dabei steht $x_i \succ x_{i+1}$ für $(x_i, x_{i+1}) \in \succ$.

Das kanonische Beispiel für eine wohlfundierte Relation ist die Ordnungsrelation $>$ auf $\mathbb{N}$. Beachten Sie, dass die Ordnungsrelation $>$ auf $\mathbb{Z}$ nicht wohlfundiert ist, da es dort beispielsweise die unendliche Kette $0 > -1 > -2 > -3 > \cdots$ gibt.

Wir demonstrieren die Technik für Terminierungsbeweise am Beispiel der Prozedur

```
fun p(n,x) = if n*n>x then n-1 else p(n+1,x)
val p : int * int -> int
```

Terminierungsbedingungen

Um die Terminierung von `p` für alle Argumente $(n, x) \in \mathbb{Z} \times \mathbb{Z}$ zu beweisen, genügt es, eine wohlfundierte Relation $\succ$ auf $\mathbb{Z} \times \mathbb{Z}$ anzugeben, für die die *Terminierungsbedingung*

$$\forall n, x \in \mathbb{Z}\colon \quad n^2 \leq x \;\Rightarrow\; (n, x) \succ (n+1, x)$$

erfüllt ist. Die Prämisse der Implikation der Terminierungsbedingung ist die negierte Bedingung des Konditionals der Prozedurdeklaration. Die Terminierungsbedingung lässt sich offensichtlich mechanisch herleiten. Dagegen erfordert die Angabe einer passenden wohlfundierten Relation $\succ$ etwas Kreativität. Für unser

Beispiel können wir die folgende Relation verwenden:

$$\forall n_1, x_1, n_2, x_2 \in \mathbb{Z}:$$
$$(n_1, x_1) \succ (n_2, x_2) \iff n_1 < n_2 \ \wedge \ x_1 = x_2 \ \wedge \ n_1^2 \leq x_1$$

Als zweites Beispiel betrachten wir die Fibonacci-Prozedur

```
fun fib n = if n<2 then n else fib(n-1) + fib(n-2)
val fib : int -> int
```

Um die Terminierung von `fib` für $n \in \mathbb{Z}$ zu zeigen, benötigen wir für jede der zwei rekursiven Anwendungen eine separate Terminierungsbedingung:

$$\forall n \in \mathbb{Z}: \ n \geq 2 \Rightarrow n \succ n-1$$
$$\forall n \in \mathbb{Z}: \ n \geq 2 \Rightarrow n \succ n-2$$

Eine passende wohlfundierte Relation ist:

$$\forall n_1, n_2 \in \mathbb{Z}: \quad n_1 \succ n_2 \iff n_1 > n_2 \geq 0$$

Terminierung der Ackermann-Prozedur

Unser nächstes Beispiel ist anspruchsvoller und kann beim ersten Lesen übersprungen werden. Wir betrachten die aus Abschnitt 7.2 bekannte Ackermann-Prozedur

```
fun am(x,y) = if x=0 then y+1
              else if y=0 then am(x-1,1)
                   else am(x-1, am(x,y-1))
val am : int * int -> int
```

Hier handelt es sich um eine Baumrekursion, die zusätzlich geschachtelt ist. Wir wollen zeigen, dass `am` für alle $(x, y) \in \mathbb{N} \times \mathbb{N}$ mit einem nicht-negativen Ergebnis terminiert. Der der äußeren Prozeduranwendung des Ausdrucks (dritte Zeile der Deklaration)

```
am(x-1, am(x,y-1))
```

entsprechende Aufruf bringt die Schwierigkeit mit sich, dass wir sein zweites Argument nicht unabhängig von der Prozedur `am` angeben können. Wir zeigen daher zuerst die folgende Aussage:

$$\forall x, y \in \mathbb{N} \ \forall z \in \mathbb{Z}: \quad \texttt{am}(x, y) \text{ terminiert mit } z \ \Rightarrow \ z \in \mathbb{N}$$

Für den Beweis nützen wir aus, dass für terminierende Prozeduraufrufe ein Rekursionsbaum existiert. Durch Induktion über die Tiefe des Rekursionsbaums für

$\mathtt{am}(x, y)$ zeigen wir, dass $z \in \mathbb{N}$ falls $x, y \in \mathbb{N}$. Für Rekursionsbäume der Tiefe 0 folgt die Behauptung aus der ersten Zeile der Deklaration. Für Rekursionsbäume mit einem Unterbaum folgt die Behauptung aus der zweiten Zeile der Deklaration mithilfe der Induktionsannahme für $\mathtt{am}(x-1, 1)$. Für Rekursionsbäume mit zwei Unterbäumen ist die dritte Zeile der Deklaration zuständig. Mit der Induktionsannahme für $\mathtt{am}(x, y-1)$ folgt, dass $\mathtt{am}(x, y-1)$ ein Ergebnis $z \in \mathbb{N}$ liefert. Also folgt mit der Induktionsannahme für $\mathtt{am}(x-1, z)$, dass insgesamt ein Ergebnis in $\mathbb{N}$ geliefert wird.

Wir können jetzt drei Bedingungen angeben, die für die Terminierung der Prozedur $\mathtt{am}$ für alle Argumente $(x, y) \in \mathbb{N} \times \mathbb{N}$ hinreichend sind:

$$\begin{aligned}
\forall x \in \mathbb{N}\colon&\quad x > 0 \Rightarrow (x, 0) \succ (x-1, 1)\\
\forall x, y \in \mathbb{N}\colon&\quad x > 0 \wedge y > 0 \Rightarrow (x, y) \succ (x, y-1)\\
\forall x, y, z \in \mathbb{N}\colon&\quad x > 0 \wedge y > 0 \Rightarrow (x, y) \succ (x-1, z)
\end{aligned}$$

Eine passende wohlfundierte Relation ist:

$$\begin{aligned}
&\forall x_1, y_1, x_2, y_2 \in \mathbb{N}\colon\\
&\quad (x_1, y_1) \succ (x_2, y_2) \iff x_1 > x_2 \vee (x_1 = x_2 \wedge y_1 > y_2)
\end{aligned}$$

Diese Relation gibt der ersten Komponente des Paares mehr Gewicht als der zweiten.[4]

7.4 Korrektheitsbeweise

Beweise, die zeigen, dass eine gegebene Prozedur eine gegebene Funktion berechnet, heißen *Korrektheitsbeweise*.

In Abschnitt 7.1 haben wir mithilfe von Induktion gezeigt, dass die Prozedur $\mathtt{sum}$ die Funktion $\lambda\,(x, y) \in \mathbb{N} \times \mathbb{Z}\,.\,x + y$ berechnet. Die folgende Proposition eröffnet eine zweite Technik für Korrektheitsbeweise, die die Induktion einspart.

Proposition 7.4.1 *Sei eine Prozedur p durch eine monomorphe Prozedurdeklaration definiert. Weiter sei f eine Funktion mit Dom $f \subseteq$ Dom f_p, die die definierende Gleichung der Prozedurdeklaration für alle Argumente in Dom f erfüllt. Dann $f \subseteq f_p$. Also berechnet p die Funktion f.*

[4] Bei der Anordnung von Wörtern in Lexika geht man übrigens analog vor: der erste Buchstabe hat mehr Gewicht als der zweite, der zweite mehr Gewicht als der dritte, und so weiter. Man spricht von *lexikografischen Ordnungen*.

Um zu zeigen, dass eine elementare Prozedur $p\colon t_1$ `->` t_2 eine Funktion f berechnet, können wir gemäß der Proposition wie folgt vorgehen:

1. Zeige, dass *Dom* f nur Werte des Typs t_1 enthält.
2. Zeige, dass p für alle $w \in$ *Dom* f terminiert.
3. Zeige, dass f die definierende Gleichung von p für alle Argumente $w \in$ *Dom* f erfüllt.

Als erstes Beispiel betrachten wir wieder die Prozedur

```
fun sum(x,y) = if x=0 then y else 1 + sum(x-1,y)
val sum : int * int -> int
```

Wir wollen zeigen, dass `sum` die Funktion

$$f \;=\; \lambda\,(x, y) \in \mathbb{N} \times \mathbb{Z}\mathbf{.}\; x + y$$

berechnet. Offensichtlich sind alle Paare $(x, y) \in \mathbb{N} \times \mathbb{Z}$ Elemente des Eingabetyps `int * int` von `sum`. Man sieht auch sofort, dass `sum` für alle $(x, y) \in \mathbb{N}\times\mathbb{Z}$ terminiert. Formal zeigt man das mit einer passenden wohlfundierte Relation. Da

$$\forall x \in \mathbb{N}\;\; \forall y \in \mathbb{Z}\colon \quad f(x, y) = \text{if } x = 0 \text{ then } y \text{ else } 1 + f(x - 1, y)$$

gilt, erfüllt f die definierende Gleichung von `sum` für alle $(x, y) \in$ *Dom* f. Also berechnet `sum` die Funktion f.

Als zweites Beispiel betrachten wir die Prozedur

```
fun p(n,x) = if n*n>x then n-1 else p(n+1,x)
val p : int * int -> int
```

und die Funktion

$$f = \lambda\,(n, x) \in \mathbb{N}^2\mathbf{.}\; \text{if } n^2 > x \text{ then } n - 1 \text{ else } \max\{\, k \in \mathbb{N} \mid k^2 \leq x \,\}$$

Wir wollen beweisen, dass die Prozedur `p` die Funktion f berechnet. In Abschnitt 7.3 haben wir bereits gezeigt, dass `p` für alle Argumente terminiert. Folglich genügt es, die Gültigkeit der folgenden Gleichung zu zeigen:

$$\forall n, x \in \mathbb{N}\colon \quad f(n, x) = \text{if } n^2 > x \text{ then } n - 1 \text{ else } f(n + 1, x)$$

Beweis Durch Fallunterscheidung. Seien $n, x \in \mathbb{N}$.

Sei $n^2 > x$. Dann folgt die Gleichung sofort aus der Definition von f.

Sei $n^2 \leq x$ *und* $(n + 1)^2 \leq x$. Wir müssen zeigen, dass $f(n, x) = f(n + 1, x)$. Das folgt sofort aus der Definition von f.

Sei $n^2 \leq x$ *und* $(n+1)^2 > x$. Wir müssen zeigen, dass $f(n,x) = f(n+1,x)$. Mithilfe der Definition von f folgt:

$$f(n,x) = \max\{\, k \in \mathbb{N} \mid k^2 \leq x \,\} = n = f(n+1)$$

□

Wir wissen jetzt, dass die Prozedur p die Funktion f berechnet. Da

$$\forall x \in \mathbb{N}\colon\ \lfloor \sqrt{x} \rfloor = \max\{\, k \in \mathbb{N} \mid k^2 \leq x \,\}$$

gilt, folgt, dass die Prozedur

```
fn x => p(0,x)
```

die Funktion

$$\lambda\, x \in \mathbb{N}\mathtt{.}\ \lfloor \sqrt{x} \rfloor$$

berechnet.

7.5 Listeninduktion

Sei X eine Menge. Für Listen über X gilt die folgende Induktionsregel:

$$\frac{A(nil) \qquad \forall x \in X\ \forall xr \in \mathcal{L}(X)\colon A(xr) \Rightarrow A(x::xr)}{\forall xs \in \mathcal{L}(X)\colon A(xs)} \tag{7.1}$$

Die durch diese Regel formulierte Beweistechnik wird als *Listeninduktion* oder als *strukturelle Induktion* bezeichnet. Mithilfe von Listeninduktion lassen sich Eigenschaften von Funktionen, deren Argumente Listen sind, bequem beweisen. Alternativ, aber weniger bequem, kann man auch Induktion über die Länge von Listen verwenden.

Sei X eine Menge. Als Ausgangspunkt für die nachfolgenden Beweise definieren

wir drei bereits bekannte Listenfunktionen wie folgt:

$$
\begin{array}{ll}
@ \in \mathcal{L}(X) \times \mathcal{L}(X) \to \mathcal{L}(X) & \text{Konkatenation} \\
\quad nil@ys = ys & \\
(x :: xr)@ys = x :: (xr@ys) &
\end{array}
$$

$$
\begin{array}{ll}
|\_| \in \mathcal{L}(X) \to \mathbb{N} & \text{Länge} \\
\quad |nil| = 0 & \\
|x :: xr| = 1 + |xr| &
\end{array}
$$

$$
\begin{array}{ll}
rev \in \mathcal{L}(X) \to \mathcal{L}(X) & \text{Reversion} \\
\quad rev(nil) = nil & \\
rev(x :: xr) = rev(xr)@[x] &
\end{array}
$$

Proposition 7.5.1 *Sei X eine Menge und seien $xs, ys, zs \in \mathcal{L}(X)$. Dann gilt:*

1. $(xs@ys)@zs = xs@(ys@zs)$ *(Assoziativität der Konkatenation)*
2. $xs@nil = xs$
3. $|xs@ys| = |xs| + |ys|$
4. $|rev(xs)| = |xs|$
5. $rev(xs@ys) = rev(ys)@rev(xs)$
6. $rev(rev(xs)) = xs$

Beweis Wir beweisen (1) durch Induktion über xs.

Sei $xs = nil$. Dann

$$
\begin{array}{lll}
(xs@ys)@zs = ys@zs & & \text{Definition von } @ \\
\quad = nil@(ys@zs) & & \text{Definition von } @ \\
\quad = xs@(ys@zs) & &
\end{array}
$$

Sei $xs = x :: xr$. Dann

$$
\begin{array}{lll}
(xs@ys)@zs = (x :: (xr@ys))@zs & & \text{Definition von } @ \\
\quad = x :: ((xr@ys)@zs) & & \text{Definition von } @ \\
\quad = x :: (xr@(ys@zs)) & & \text{Induktionsannahme} \\
\quad = xs@(ys@zs) & & \text{Definition von } @
\end{array}
$$

Die anderen Behauptungen lassen sich analog zeigen. Für den Beweis von (4) benötigt man (3). Für den Beweis von (5) benötigt man (1) und (2). Für den Beweis von (6) benötigt man (5). □

Die Definitionen der Funktionen @, |_| und *rev* entsprechen den Prozedurdeklarationen für `append`, `length` und `reverse` in Kapitel 4. Allerdings handelt es sich um polymorphe Prozeduren. Wenn wir uns aber auf eine Typ t für die Typvariable festlegen, und X als die Menge aller Werte von t wählen, bekommen wir monomorphe Prozeduren, denen genau die obigen Funktionen zugeordnet sind. Also gelten die in Proposition 7.5.1 formulierten Eigenschaften auch für die Prozeduren.

7.6 Laufzeiten

Definition 7.6.1 *Die* Laufzeit eines terminierenden Prozeduraufrufs *A ist die Anzahl aller Aufrufe von rekursiven Prozeduren, die für die Ausführung von A ausgeführt werden müssen. Die* Laufzeit einer Prozedur *ist die Funktion, die den Argumenten, für die die Prozedur termininiert, die Laufzeit des entsprechenden Aufrufs zuordnet.*

Damit haben wir die Laufzeit einer Prozedur unabhängig davon definiert, mit welchem Computer und mit welchem Interpreter sie ausgeführt wird. Es spielt auch keine Rolle, ob die Prozedur in einer Programmiersprache oder mit mathematischer Notation geschrieben ist.

Unsere Definition der Laufzeit zählt nur die Aufrufe von rekursiven Prozeduren. Alternativ kann man auch die Aufrufe aller Prozeduren und aller Operationen (zum Beispiel $4 + 2$) zählen. Ein solche Definition würde der echten Laufzeit auf einem Interpreter besser entsprechen. Da sie aber für die wesentliche Aussagen, die wir in diesem Kapitel machen wollen (asymptotische Laufzeiten, siehe Abschnitt 7.8) keinen Unterschied macht, haben wir die Definition gewählt, die am wenigsten Zählerei erfordert.

Sei A ein terminierender Prozeduraufruf. Wir betrachten zwei Spezialfälle:

1. Wenn für die Ausführung von A keine rekursive Prozedur aufgerufen werden muss, dann hat A gemäß unserer Definition die Laufzeit 0.

2. Wenn A der Aufruf einer rekursiven Prozedur ist, die keine anderen rekursiven Prozeduren aufruft, dann ist Laufzeit von A gleich der Größe des Rekursionsbaums von A.

Die in Abschnitt 7.2 gezeigten Rekursionsbäume für die Fibonacci- und Ackermann-Prozedur legen nahe, dass diese Prozeduren hohe Laufzeiten haben. In der Tat benötigt der Prozeduraufruf

```
fib 40
102334155 : int
```

mehrere Sekunden. Die Laufzeit der Ackermann-Prozedur ist noch gigantischer. Der Aufruf

```
am(4,4)
```

liefert auch nach einigen Minuten noch kein Ergebnis (obwohl er terminiert, siehe Abschnitt 7.3).

Oft gelingt es, die Laufzeit einer Prozedur explizit anzugeben. Unser erstes Beispiel ist die Fakultätsprozedur:

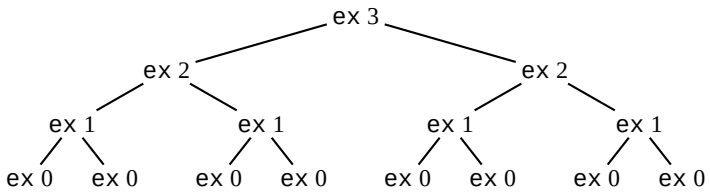
```
fun fak n = if n=0 then 1 else n*fak(n-1)
```

Man sieht sofort, dass `fak` für $n \in \mathbb{N}$ terminiert. Der Rekursionsbaum für einen Aufruf `fak n` ist linear und hat die Tiefe n. Also hat `fak` für $n \in \mathbb{N}$ die Laufzeit $n+1$.

Unser zweites Beispiel ist die Prozedur[5]

```
fun ex n = if n<1 then () else (ex(n-1) ; ex(n-1))
```

Diese hat für 3 den folgenden Rekursionsbaum:



Für $n \in \mathbb{N}$ ist der Rekursionsbaum für `ex` n ein balancierter Binärbaum der Tiefe n. Also hat der Rekursionsbaum für `ex` n die Größe $2^{n+1}-1$ (Proposition 5.4.4). Folglich hat `ex` n für $n \in \mathbb{N}$ die Laufzeit $2^{n+1}-1$.

Die Prozedur `ex` hat also schon für relativ kleine n eine gigantische Laufzeit. Nehmen Sie an, dass Sie einen Computer haben, der pro Sekunde 10^{10} Prozedu-

[5] Diese Prozedur terminiert für alle Argumente und liefert immer das triviale Ergebnis (). Sie dient nur dazu, den Interpreter richtig zum Arbeiten zu bringen.

raufrufe ausführen kann.[6] Um `ex` 400 zu berechnen, benötigen wir dann

$$\frac{2^{400}}{10^{10} \cdot 3600 \cdot 24 \cdot 366} \geq \frac{16^{100}}{10^{10} \cdot 10^4 \cdot 10^2 \cdot 10^3} \geq \frac{10^{100}}{10^{19}} = 10^{81}$$

mindestens 10^{81} Jahre. Das ist ziemlich lange.

Unser drittes Beispiel ist die Konkatenationsprozedur für Listen:

```
fun append (nil  , ys) = ys
  | append (x::xr, ys) = x::append(xr,ys)
val append : 'a list * 'a list -> 'a list
```

Da das erste Argument bei jedem rekursiven Aufruf kürzer wird, terminiert `append` für alle Argumente. Der Rekursionsbaum für einen Aufruf `append`(xs, ys) hat offensichtlich die Tiefe $|xs|$. Also hat `append`(xs, ys) die Laufzeit $|xs| + 1$.

Naives Reversieren

Unser viertes Beispiel ist die sogenannte *naive Reversionsprozedur*:

```
fun reverse nil     = nil
  | reverse (x::xr) = append(reverse xr, [x])
val reverse : 'a list -> 'a list
```

Man sieht sofort, dass `reverse` für alle Argumente terminiert. Der Rekursionsbaum für einen Aufruf `reverse` *xs* hat offensichtlich die Tiefe $|xs|$. Die Größe des Rekursionsbaums liefert uns diesmal allerdings nicht die Laufzeit, da wir die Aufrufe der rekursiven Hilfsprozedur `append` mitzählen müssen.

Wir wählen einen Elementtyp t für die Listen. Offensichtlich hängt die Laufzeit von `reverse` nicht von der Wahl von t ab. Sei X die Menge aller Werte des Typs t. Die in Abschnitt 7.5 für X definierten Funktionen @ und *rev* entsprechen offensichtlich genau den Prozeduren `append` und `reverse`, wenn wir diese auf Listen über t einschränken. Also wissen wir mit Proposition 7.5.1, dass `reverse` *xr* eine Liste der Länge $|xr|$ liefert.

Sei ϕ die Laufzeit von `reverse`. Offensichtlich gilt für alle $x \in X$ und $xr \in \mathcal{L}(X)$:

$$\begin{aligned} \phi(nil) &= 1 \\ \phi(x::xr) &= 1 + \phi(xr) + (|xr| + 1) = 2 + \phi(xr) + |xr| \end{aligned}$$

[6] Das ist sehr optimistisch, da ein n MHz schneller Computer höchsten $n \cdot 10^6$ Prozeduraufrufe pro Sekunde ausführen kann.

Dabei geht die Laufzeit von `append`$(rev(xr),\ [x])$ mit $|xr|+1$ ein. Die obigen Gleichungen werden als *Laufzeitgleichungen* bezeichnet.

Aus den Laufzeitgleichungen kann man sofort sehen, dass ϕ nur von der Länge der Argumentliste abhängt. Mit dieser Einsicht können wir die Laufzeitgleichungen zu einer Prozedur

```
fun phi n = if n=0 then 1 else 2 + phi(n-1) + (n-1)
val phi : int -> int
```

umformulieren, die die Laufzeit von `reverse` für Listen der Länge n berechnet.

Durch Induktion über *xs* kann man zeigen:

$$\forall xs \in \mathcal{L}(X):\ \phi(xs) = \frac{1}{2}|xs|^2 + \frac{3}{2}|xs| + 1 \tag{7.2}$$

Beweis *Sei xs = nil.* Dann folgt mit der ersten Laufzeitgleichung:

$$\phi(xs) = 1 = \frac{1}{2}|xs|^2 + \frac{3}{2}|xs| + 1$$

Sei xs = x::nil. Dann folgt mit der zweiten Laufzeitgleichung:

$$\begin{aligned}\phi(xs) &= 2 + \phi(xr) + |xr| \\ &= 2 + \frac{1}{2}|xr|^2 + \frac{3}{2}|xr| + 1 + |xr| && \text{Induktionsannahme} \\ &= 2 + \frac{1}{2}(|xs|-1)^2 + \frac{3}{2}(|xs|-1) + |xs| \\ &= \frac{1}{2}|xs|^2 + \frac{3}{2}|xs| + 1\end{aligned}$$

□

Gleichung 7.2 haben wir aus dem Hut gezaubert. Mit etwas Übung ersieht man aus den Laufzeitgleichungen, dass

$$\forall xs \in \mathcal{L}(X):\ \phi(xs) = \sum_{n=0}^{|xs|}(1+n)$$

gilt. Wenn man die Summe ausrechnet,

$$\begin{aligned}\sum_{n=0}^{|xs|}(1+n) &= |xs| + 1 + \sum_{n=1}^{|xs|} n \\ &= |xs| + 1 + \frac{|xs|}{2}(|xs|+1) \\ &= \frac{1}{2}|xs|^2 + \frac{3}{2}|xs| + 1\end{aligned}$$

bekommt man Gleichung 7.2.

7.7 Die Funktionsklassen O und Theta

Für praktische Belange genügt es, die Laufzeiten von Prozeduren approximativ zu bestimmen. Dazu verwendet man sogenannte Funktionsklassen. Unter einer *Funktionsklasse* verstehen wir hier eine beliebige Teilmenge von $\mathbb{N} \to \mathbb{R}$.

Definition 7.7.1 *Seien $f, g \in \mathbb{N} \to \mathbb{R}$. Wir sagen „$f$* dominiert *$g$“ und schreiben $g \preceq f$ genau dann, wenn*

$$\exists n_0 \in \mathbb{N}\ \ \exists c \in \mathbb{N}^+\ \ \forall n \geq n_0 \colon\ \ g(n) \leq c \cdot f(n)$$

Für alle $f, g, h \in \mathbb{N} \to \mathbb{R}$ gilt offensichtlich:

- $f \preceq f$ (Reflexivität von $\preceq$)
- $f \preceq g \wedge g \preceq h \ \Rightarrow\ f \preceq h$ (Transitivität von $\preceq$)

Für jede Funktion $f \in \mathbb{N} \to \mathbb{R}$ definieren wir zwei Funktionsklassen:

$$O(f) = \{\, g \in \mathbb{N} \to \mathbb{R} \mid g \preceq f \,\} \qquad \text{(sprich „O } f\text{“)}$$

$$\Theta(f) = \{\, g \in \mathbb{N} \to \mathbb{R} \mid f \preceq g \wedge g \preceq f \,\} \qquad \text{(sprich „Theta } f\text{“)}$$

Offensichtlich gilt für alle $f, g \in \mathbb{N} \to \mathbb{R}$:

1. $\Theta(f) \subseteq O(f)$
2. $O(f) = O(g) \iff \Theta(f) = \Theta(g)$

Hier ist ein Beispiel:

$$(\lambda\, n \in \mathbb{N}\,.\, 7n^2 - 5n + 13) \in \Theta(\lambda\, n \in \mathbb{N}\,.\, n^2)$$

Da die Lambda-Präfixe die Lesbarkeit erschweren und aus dem Kontext leicht ableitbar sind, lässt man sie üblicherweise weg und schreibt kürzer

$$7n^2 - 5n + 13 \in O(n^2)$$

Wir definieren für $n \in \mathbb{N}$:

$$\log n \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } 0 \text{ else } \log_2 n$$

Da für $a, b, x > 0$

$$\log_a x = \log_a b \cdot \log_b x$$

gilt, haben wir für alle $a > 1$:

$$\Theta(\log n) = \Theta(\text{if } n = 0 \text{ then } 0 \text{ else } \log_a n)$$

Proposition 7.7.2

1. $O(n \log n) \subsetneq O(n^a) \subsetneq O(n^b)$ *falls* $1 < a < b$
2. $O(n^a) \subsetneq O(b^n) \subsetneq O(c^n)$ *falls* $1 < a$ *und* $1 < b < c$

Proposition 7.7.3 *Die Funktionsklassen*

$$O(1),\quad \Theta(\log n),\quad \Theta(n),\quad \Theta(n \log n),\quad \Theta(n^2),\quad \Theta(n^3),\quad \Theta(2^n)$$

sind disjunkt. Außerdem gilt:

$$O(1) \subsetneq O(\log n) \subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq O(2^n)$$

Bei der Beschreibung der Klassen $O(f)$ und $\Theta(f)$ kann man Details radikal weglassen. Beispielsweise gilt:

$$\begin{aligned} O(897) &= O(1) \\ \theta(67 \log n + 33) &= \Theta(\log n) \\ \Theta(7n + \log n + 45) &= \Theta(n) \\ \Theta(56n \log n + 67n + 45) &= \Theta(n \log n) \\ \Theta(\frac{n^2}{7} + 789n \log n + 45n + 77) &= \Theta(n^2) \\ \Theta(\frac{2^n}{23} + 12n^2 + 789n + 77) &= \Theta(2^n) \end{aligned}$$

7.8 Asymptotische Laufzeiten

Zunächst betrachten wir nur Prozeduren mit dem Eingabetyp `int` und interessieren uns nur für die Laufzeiten bei nicht-negativen Argumenten.

Laufzeit $\Theta(f)$
Laufzeit $O(f)$

Wir sagen, dass eine Prozedur p: `int` $\to t$ die *Laufzeit* $\Theta(f)$ hat genau dann, wenn es ein $n_0 \in \mathbb{N}$ und ein $g \in \Theta(f)$ gibt, sodass für alle $n \geq n_0$ der Aufruf $p\ n$ die Laufzeit $g(n)$ hat. Entsprechend sagen wir, dass p die *Laufzeit* $O(f)$ hat genau dann, wenn es ein $n_0 \in \mathbb{N}$ und ein $g \in O(f)$ gibt, sodass für alle $n \geq n_0$ der Aufruf $p\ n$ die Laufzeit $g(n)$ hat.

Wenn eine Prozedur p die Laufzeit $\Theta(n^2)$ hat, sagt man auch kürzer, dass p *quadratische Laufzeit* hat. Wenn p die Laufzeit $O(n^2)$ hat, sagt man entsprechend, dass p *höchstens quadratische Laufzeit* hat. Entsprechend vereinbaren wir die folgenden Sprechweisen:

$O(1)$	konstante Laufzeit
$\Theta(\log n)$	logarithmische Laufzeit
$\Theta(n)$	lineare Laufzeit
$\Theta(n^2)$	quadratische Laufzeit
$\Theta(n^3)$	kubische Laufzeit
$\Theta(a^n)$	exponentielle Laufzeit (zur Basis $a > 1$)

Laufzeitangaben mit O und Θ bezeichnet man als *asymptotische Laufzeitangaben*. Laufzeitangaben ohne O und Θ bezeichnet man dagegen als *exakte Laufzeitangaben*. Die exakten Laufzeitaussagen aus Abschnitt 7.6 können wir wie folgt asymptotisch formulieren:

1. Die Fakultätsprozedur `fak` hat für $n \in \mathbb{N}$ lineare Laufzeit ($\Theta(n)$).
2. Die Prozedur `ex` hat für $n \in \mathbb{N}$ exponentielle Laufzeit ($\Theta(2^n)$).
3. Die Konkatenationsprozedur `append` hat für Argumente (xs, ys) lineare Laufzeit bezüglich $|xs|$ ($\Theta(|xs|)$).
4. Die naive Reversionsprozedur `reverse` hat für Argumente xs quadratische Laufzeit bezüglich $|xs|$ ($\Theta(|xs|^2)$).

In der Praxis haben die meisten Prozeduren lineare Laufzeit. Prozeduren mit logarithmischer Laufzeit sind besonders schnell. Die Laufzeit $\Theta(n \log n)$ ist in der Praxis kaum von linearer Laufzeit unterscheidbar. Quadratische Laufzeit ist in der Praxis oft schon zu langsam (Probieren Sie die naive Reversionsprozedur für eine Liste der Länge 50000, siehe Abschnitt 4.10). Exponentielle Laufzeit ist fast immer zu langsam. Wenn man eine Prozedur schreibt, sollte man grundsätzlich auf ihre asymptotische Laufzeit achten. Prozeduren mit quadratischer oder schlechterer Laufzeit sind Kandidaten für Performanzprobleme.

Oft ist es möglich, eine langsame Prozedur durch eine schnellere Prozedur zu ersetzen. In den folgenden Abschnitten werden wir dafür einige Beispiele sehen.

Wir haben bereits asymptotische Laufzeitangaben für zwei Prozeduren gemacht, deren Argumente keine natürlichen Zahlen sind (`append` und `reverse`). Wir wollen uns jetzt genauer ansehen, wie man dabei vorgeht. Als Beispiel betrachten wir die Prozedur

```
fun f(x,y) = if x=0 orelse y=0 then () else f(x-1,y-1)
val f : int * int -> unit
```

Die Laufzeit dieser Prozedur für Argumente $(x, y) \in \mathbb{N}^2$ ist

$$\phi = \lambda\,(x, y) \in \mathbb{N}^2 \,.\, \min\{x, y\} + 1$$

Die Laufzeitfunktion ϕ zerlegen wir in eine *Größenfunktion*

$$\gamma = \lambda\,(x, y) \in \mathbb{N}^2\,.\,\min\{x, y\}$$

und in eine *Aufwandsfunktion*

$$\alpha = \lambda\, n \in \mathbb{N}\,.\,n + 1$$

mit

$$\phi = \lambda\, a \in \mathbb{N}^2\,.\,\alpha(\gamma(a))$$

Die Klasse $\Theta(\alpha) = \Theta(n)$ wird dann als asymptotische Laufzeit von `f` bezüglich der Größenfunktion γ für Argumente in $\mathbb{N}^2$ bezeichnet. In der Praxis gibt man die Zerlegung in Größen- und Aufwandsfunktion meistens nicht explizit an. Stattdessen sagt man einfacher, dass `f` für Argumente $(x, y) \in \mathbb{N}^2$ die Laufzeit $\Theta(\min\{x, y\})$ hat.

Im Allgemeinen gibt es jedoch ein kleines Problem: Es kann sein, dass die betrachtete Prozedur für verschiedene Argumente mit der gleichen Größe (gemäß der Größenfunktion) verschiedene Laufzeiten hat. In diesem Fall definiert man die Aufwandsfunktion als das Maximum der für Argumente einer Größe möglichen Laufzeiten. Die Zerlegung in Größen- und Aufwandsfunktion ist nur dann zulässig, wenn dieses Maximum auch für alle Größen existiert.

Allgemeiner kann man zu einer Größenfunktion eine *obere Aufwandsfunktion* (Maximum der möglichen Laufzeiten, „worst case“) und eine *untere Aufwandsfunktion* (Minimum der möglichen Laufzeiten, „best case“) definieren. Eine Θ-Abschätzung des Aufwands ist nur dann möglich, wenn die untere und die obere Aufwandsfunktion in derselben Θ-Klasse liegen. Wenn das nicht der Fall ist, gibt man sich in der Regel mit einer Θ-Abschätzung der oberen Aufwandsfunktion zufrieden. Diese bezeichnet man dann als *scharfe O-Laufzeit*.

scharfe O-Laufzeit

Als Beispiel betrachten wir die Prozedur

```
fun m(x,y) = if x<y then x else m(x-y,y)
val m : int * int -> int
```

Wir interessieren uns nur für Argumente in $\mathbb{N} \times \mathbb{N}^+$. Als Größenfunktion wählen wir

$$\lambda\,(x, y) \in \mathbb{N} \times \mathbb{N}^+\,.\,x$$

Die obere Aufwandsfunktion ist dann in $\Theta(x)$ und die untere Aufwandsfunktion in $\Theta(1)$. Man sagt, dass $O(x)$ eine *scharfe obere Schranke* für die Laufzeit von `m` für Argumente $(x, y) \in \mathbb{N} \times \mathbb{N}^+$ ist.

7.9 Schnelle Exponentiation

Wir wollen eine Prozedur schreiben, die zu $x \in \mathbb{Z}$ und $n \in \mathbb{N}$ die Potenz x^n mit der Laufzeit $\Theta(\log n)$ berechnet. Um die logarithmische Laufzeit zu erhalten, müssen wir das Argument n bei jedem Rekursionsschritt halbieren. Die Gleichungen

$$\begin{aligned} x^{2n} &= (x \cdot x)^n \\ x^{2n+1} &= x \cdot (x \cdot x)^n \end{aligned}$$

liefern die folgende Prozedur:

```
fun exp(x,n) = if n = 0 then 1
               else if n mod 2 = 0 then exp(x*x, n div 2)
                    else x * exp(x*x, n div 2)

val exp : int * int -> int
```

Die Terminierung von `exp` für $(x, n) \in \mathbb{Z} \times \mathbb{N}$ ist offensichtlich. Da die Funktion

$$\lambda\,(x, n) \in \mathbb{Z} \times \mathbb{N}\,.\,x^n$$

die definierende Gleichung der Prozedur `exp` für alle $(x, n) \in \mathbb{Z} \times \mathbb{N}$ erfüllt (siehe die obigen Gleichungen), berechnet `exp` die Funktion $\lambda\,(x, n) \in \mathbb{Z} \times \mathbb{N}\,.\,x^n$ (Propositon 7.4.1).

Proposition 7.9.1 *Für* $(x, n) \in \mathbb{Z} \times \mathbb{N}$ *hat* `exp` *die Laufzeit* $\Theta(\log n)$.

Beweis Sei $\phi(x, n)$ die Laufzeit von `exp`(x, n) für $(x, n) \in \mathbb{Z} \times \mathbb{N}$. Wir zeigen durch Induktion über n:

$$\forall n \in \mathbb{N}\ \forall x \in \mathbb{Z}\colon\ \phi(x, n) = 2 + (\text{if } n = 0 \text{ then } -1 \text{ else } \lfloor \log_2 n \rfloor)$$

Sei $n = 0$. Dann $\phi(x, n) = 1$.

Sei $n = 1$. Dann $\phi(x, n) = 1 + \phi(x, 0) = 2$.

Sei $n > 1$. Dann

$$\begin{aligned} \phi(x, n) &= 1 + \phi(x, \lfloor n/2 \rfloor) && \text{Definition } \texttt{exp} \\ &= 3 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor && \text{Induktionsannahme} \\ &= 3 + \lfloor \log_2 (n/2) \rfloor \\ &= 3 + \lfloor \log_2 n - 1 \rfloor \\ &= 2 + \lfloor \log_2 n \rfloor \end{aligned}$$

□

7.10 Größte gemeinsame Teiler

Die Prozedur

```
fun gcd(x,y) = if y=0 then x else gcd(y, x mod y)
val gcd : int * int -> int
```

realisiert den *Euklidischen Algorithmus*, der schon im Altertum bekannt war. Wir werden zeigen, dass `gcd` für $(x, y) \in \mathbb{N}^2$ den größten gemeinsamen Teiler von x und y mit der Laufzeit $O(\log y)$ berechnet.[7]

Korrektheit von gcd

Als Grundlage für den Korrektheitsbeweis müssen wir zuerst definieren, was der größte gemeinsame Teiler von zwei natürlichen Zahlen ist.

Die *Teiler* einer Zahl $x \in \mathbb{N}$ sind wie folgt definiert:

$$T(x) = \{\, k \in \mathbb{N}^+ \mid \exists n \in \mathbb{N}\colon\ x = k \cdot n \,\}$$

Sei $k \in \mathbb{N}^+$ und $x \in \mathbb{N}$. Dann gilt:

$$k \in T(x) \iff \exists n \in \mathbb{N}\colon\ x = k \cdot n \iff x = \left\lfloor \frac{x}{k} \right\rfloor k \iff x \bmod k = 0$$

Weiter gilt $T(0) = \mathbb{N}^+$. Für $x > 0$ gilt $\{1, x\} \subseteq T(x) \subseteq \{1, \ldots, x\}$.

Der größte gemeinsame Teiler zweier natürlicher Zahlen ist wie folgt definiert:

$$ggt \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$ggt(x, y) = \text{if } x = 0 \wedge y = 0 \text{ then } 0 \text{ else } \max(T(x) \cap T(y))$$

Wir werden beweisen, dass die Prozedur `gcd` die Funktion *ggt* berechnet.

Zuerst überlegen wir uns, dass `gcd` für $(x, y) \in \mathbb{N}^2$ terminiert. Das folgt sofort aus der Tatsache:

$$\forall x, y \in \mathbb{N}\colon\ y \neq 0 \Rightarrow 0 \leq x \bmod y < y$$

Wegen Proposition 7.4.1 genügt es nun zu zeigen, dass *ggt* die definierende Gleichung von `gcd` erfüllt. Dies ist offensichtlich der Fall, wenn die folgenden zwei Gleichungen gelten:

$$\forall x \in \mathbb{N}\colon\ ggt(x, 0) = x$$
$$\forall x \in \mathbb{N}\ \forall y \in \mathbb{N}^+\colon\ T(x) \cap T(y) = T(y) \cap T(x \bmod y)$$

Die erste Gleichung rechnet man leicht nach. Die zweite Gleichung beweisen wir.

[7] gcd steht für „greatest common divisor“.

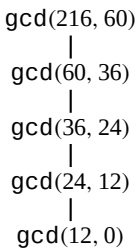
Beweis Sei $x \in \mathbb{N}$ und $y \in \mathbb{N}^+$.

Sei $k \in T(x) \cap T(y)$. Also gilt $x = \lfloor x/k \rfloor k$ und $y = \lfloor y/k \rfloor k$. Da $0 \leq x \bmod y = x - \lfloor x/y \rfloor y = k \cdot (\lfloor x/k \rfloor - \lfloor x/y \rfloor \cdot \lfloor y/k \rfloor)$, gilt $k \in T(x \bmod y)$. Also $k \in T(y) \cap T(x \bmod y)$.

Sei $k \in T(y) \cap T(x \bmod y)$. Also gilt $y = \lfloor y/k \rfloor k$ und $x \bmod y = \lfloor (x \bmod y)/k \rfloor k$. Da $x = \lfloor x/y \rfloor y + x \bmod y$, gilt $k \in T(x)$. Also $k \in T(x) \cap T(y)$. □

Laufzeit von gcd

Für $x \in \mathbb{N}$ hat $\texttt{gcd}(x, x)$ offensichtlich die Laufzeit 1 oder 2. Wenn die beiden Argumente verschieden sind, können die Rekursionsbäume tiefer werden. Hier ist ein Beispiel:



Die entscheidende Beobachtung für die Laufzeitabschätzung der Prozedur gcd ist die Tatsache, dass das zweite Argument nach zwei Rekursionschritten höchstens noch halb so groß wie zu Anfang ist.

Lemma 7.10.1 *Seien* $x \in \mathbb{N}$, $y \in \mathbb{N}^+$ *und* $x \bmod y \geq 1$. *Dann*

$$y \bmod (x \bmod y) < \frac{y}{2}$$

Beweis

$$\begin{aligned} y &= \lfloor y/x \bmod y \rfloor \, (x \bmod y) + y \bmod (x \bmod y) \\ &\geq x \bmod y + y \bmod (x \bmod y) && \text{da } y > x \bmod y \\ &> y \bmod (x \bmod y) + y \bmod (x \bmod y) && \text{da } x \bmod y > y \bmod (x \bmod y) \\ &= 2(y \bmod (x \bmod y)) \end{aligned}$$

□

Proposition 7.10.2 *Die Prozedur* gcd *hat für* $(x, y) \in \mathbb{N}^2$ *die Laufzeit* $O(\log y)$.

Beweis Sei $\phi(x, y)$ die Laufzeit von $\texttt{gcd}(x, y)$ für $x, y \in \mathbb{N}$. Wir beweisen durch Induktion über y:

$$\forall x, y \in \mathbb{N}\colon\ \phi(x, y) \leq 2 + 2(\text{if } y = 0 \text{ then } 0 \text{ else } \log_2 y)$$

Sei $y = 0$ *oder* $y = 1$. Dann $\phi(x, y) \leq 2$.

Sei $y \geq 2$. Wir unterscheiden drei Fälle:

1. $x \bmod y = 0$. Dann $\phi(x, y) = 2 < 2 + 2\log_2 y$.
2. $x \bmod y > 0$ *und* $y \bmod (x \bmod y) = 0$. Dann $\phi(x, y) = 3 < 2 + 2\log_2 y$.
3. $x \bmod y > 0$ *und* $y \bmod (x \bmod y) > 0$. Dann

$$\begin{aligned}
\phi(x, y) &= 1 + \phi(y,\ x \bmod y) && \text{Definition } \texttt{gcd}\\
&= 2 + \phi(x \bmod y,\ y \bmod (x \bmod y)) && \text{Definition } \texttt{gcd}\\
&\leq 4 + 2\log_2(y \bmod (x \bmod y)) && \text{Induktionsannahme}\\
&< 4 + 2\log_2 \frac{y}{2} && \text{Lemma 7.10.1}\\
&= 2 + 2\log_2 y
\end{aligned}$$

□

7.11 Iterative Rekursion

Iterative Rekursion ist ein wichtiger Spezialfall von linearer Rekursion. Hier ist ein Beispiel für eine iterative Rekursion:

```
fun sumi(x,y) = if x=0 then y else sumi(x-1,y+1)
```

Iterative Rekursion liegt dann vor, wenn die rekursiven Anwendungen der Prozedur das Ergebnis der Prozedur liefern (so wie bei `sumi`). Die Prozedur

```
fun sum(x,y) = if x=0 then y else 1+sum(x-1,y)
```

ist also nicht iterativ-rekursiv. Offensichtlich berechnen `sumi` und `sum` beide die Funktion $\lambda\,(x, y) \in \mathbb{N} \times \mathbb{Z}.\ x + y$.

Statt von iterativer Rekursion spricht man auch von *Endrekursion* oder kurz von *Iteration*. Statt iterativ-rekursiv sagt man kürzer *iterativ*.

Iterative Prozeduren haben gegenüber nicht-iterativen Prozeduren den praktischen Vorteil, dass ihre Ausführung im Speicher weniger Platz benötigt. Das liegt daran, dass bei nicht-iterativer Rekursion nach der Ausführung eines rekursiven Aufrufs an die aufrufende Stelle zurückkehrt werden muss, diese also im Speicher vermerkt werden muss. Dazu benötigt man Platz proportional zur Tiefe des Rekursionsbaums. Bei iterativen Berechnungen entfällt diese schrittweise Rückkehr, da man vom Blatt des Rekursionsbaums direkt zur Wurzel zurückkehren kann (iterative Rekursionsbäume sind immer linear, es gibt also genau ein Blatt).

Den unterschiedlichen Platzbedarf von iterativer und nicht-iterativer Rekursion kann man auch mithilfe von Auswertungsprotokollen erklären. Dazu betrachten wir zwei gekürzte Ausführungsprotokolle für `sumi` und `sum`:

```
sumi(4,6)  →+  sumi(3,7)
           →+  sumi(2,8)
           →+  sumi(1,9)
           →+  sumi(0,10)
           →+  10

sum(4,6)   →+  1 + sum(3,6)
           →+  1 + (1 + sum(2,6))
           →+  1 + (1 + (1 + sum(1,6)))
           →+  1 + (1 + (1 + (1 + sum(0,6))))
           →+  10
```

Mit zunehmender Rekursionstiefe werden bei `sum` also im Gegensatz zu `sumi` immer größere Zwischenausdrücke gebildet.

Dass `sum` mit steigender Rekursionstiefe im Gegensatz zu `sumi` mehr und mehr Speicherplatz benötigt, kann experimentell wie folgt beobachtet werden:

```
sum(1000000,0)
! Uncaught exception: Out_of_memory

sumi(1000000,0)
100000 : int
```

Die iterative Prozedur `sumi` rechnet auch schneller als ihre nicht-iterative Schwester `sum`, da der Verwaltungsaufwand für die wiederholte Rückkehr zum jeweils übergeordneten Aufruf entfällt. Allerdings ist dieser konkrete Laufzeitunterschied in unserem abstrakten Laufzeitmodell nicht sichtbar, da beide Prozeduren mit der gleichen Rekursionstiefe rechnen.

Iterative Rekursion kann auch mit kaskadierten Prozeduren realisiert werden:

```
fun sumi x y = if x=0 then y else sumi (x-1) (y+1)
```

Der Interpreter behandelt die kaskadierte Anwendung

```
sumi (x-1) (y+1)
```

übrigens wie eine kartesische Anwendung, da er die Berechnung der Hilfsprozedur `sumi (x-1)` einspart.

Praktische Konsequenzen

Wenn mit größerer Rekursionstiefe gerechnet werden soll, sollte man grundsätzlich mit iterativen Prozeduren arbeiten.

In Abschnitt 4.5 haben wir zwei Faltungsprozeduren `foldl` und `foldr` kennengelernt. Während `foldl` iterativ ist, ist dies für `foldr` nicht der Fall. Wenn man die Wahl zwischen `foldl` und `foldr` hat, sollte man daher grundsätzlich die iterative Prozedur `foldl` benutzen.

7.12 Einführung von Akkumulatorargumenten

Oft können linear-rekursive Prozeduren durch die Einführung von sogenannten *Akkumulatorargumenten* in iterative Form gebracht werden. Als Beispiel betrachten wir die Fakultätsprozedur:[8]

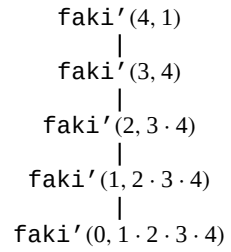
```
fun fak n = if n=0 then 1 else n*fak(n-1)
```

Für die iterative Version `faki` benötigen wir eine Hilfprozedur `faki'` mit einem zusätzlichen Akkumulatorargument:

```
fun faki'(n,a) = if n=0 then a else faki'(n-1, n*a)

fun faki n = faki'(n,1)
```

Um die Arbeitsweise von `faki'` zu verstehen, betrachten wir einen Rekursionsbaum:



Wir wollen jetzt zeigen, dass `fak` und `faki` für $n \in \mathbb{N}$ die gleiche Funktion berechnen. Zuerst beweisen wir eine Eigenschaft von *faki'*:[9]

$$\forall n \in \mathbb{N}\ \forall a \in \mathbb{N}\colon\ \mathit{faki}'(n, a) = a \cdot \mathit{faki}'(n, 1) \tag{7.3}$$

Beweis Durch Induktion über n.

Sei $n = 0$. Dann folgt aus der Definition von *faki'*:

$$\mathit{faki}'(n, a) = a = a \cdot \mathit{faki}'(n, 1)$$

[8] Eine iterative Version der Fakultätsprozedur bringt aus praktischer Sicht keine Vorteile, da wegen der Größenbeschränkung für ganze Zahlen große Rekursionstiefen sowieso unmöglich sind.

[9] *faki'* ist die der Prozedur `faki'` zugeordnete Funktion.

Sei $n > 0$. Dann

$$\begin{aligned} \mathit{faki}'(n, a) &= \mathit{faki}'(n-1, n \cdot a) && \text{Definition } \mathit{faki}' \\ &= n \cdot a \cdot \mathit{faki}'(n-1, 1) && \text{Induktionsannahme} \\ &= a \cdot \mathit{faki}'(n-1, n) && \text{Induktionsannahme} \\ &= a \cdot \mathit{faki}'(n, 1) && \text{Definition } \mathit{faki}' \end{aligned}$$

□

Als Nächstes zeigen wir

$$\forall n \in \mathbb{N}\colon\ \mathit{fak}(n) = \mathit{faki}'(n, 1) \tag{7.4}$$

Beweis Durch Induktion über n.

Sei $n = 0$. Dann $\mathit{fak}(n) = 1 = \mathit{faki}'(n, 1)$.

Sei $n > 0$. Dann

$$\begin{aligned} \mathit{fak}(n) &= n \cdot \mathit{fak}(n-1) && \text{Definition } \mathit{fak} \\ &= n \cdot \mathit{faki}'(n-1, 1) && \text{Induktionsannahme} \\ &= \mathit{faki}'(n-1, n) && \text{wegen (7.3)} \\ &= \mathit{faki}'(n, 1) && \text{Definition } \mathit{faki}' \end{aligned}$$

□

Damit haben wir bewiesen, dass die Prozeduren `fak` und `faki` für alle Argumente $n \in \mathbb{N}$ dieselben Ergebnisse berechnen.

Iterative Listenreversion mit linearer Laufzeit

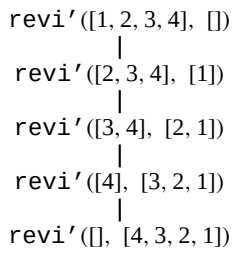
Betrachten Sie die Prozeduren

```
fun revi' (nil,   ys) = ys
  | revi' (x::xr, ys) = revi'(xr, x::ys)

fun revi xs = revi'(xs,nil)

val revi : 'a list -> 'a list
```

Um die iterative Prozedur `revi'` zu verstehen, betrachten wir den Rekursionsbaum



Offensichtlich reversiert `revi` Listen mit linearer Laufzeit bezüglich der Listenlänge. Wir beweisen den Korrektheitsteil dieser Aussage mit Listeninduktion.

Proposition 7.12.1 *Sei t ein Typ und X die Menge der Werte von t. Weiter sei $revi' \in \mathcal{L}(X) \times \mathcal{L}(X) \to \mathcal{L}(X)$ die der Prozedur* `revi'` *für den Typ t zugeordnete Funktion. Schließlich sei rev die in Abschnitt 7.5 für X definierte Funktion. Dann gilt:*

$$\forall xs, ys \in \mathcal{L}(X)\ \forall n \in \mathbb{Z}:\ revi'(xs, ys) = rev(xs)@ys$$

Beweis Durch Induktion über *xs*.

Sei $xs = nil$. Dann

$$\begin{aligned} revi'(xs, ys) &= ys && \text{Definition von } revi' \\ &= rev(nil)@ys && \text{Definition von } rev \text{ und } @ \\ &= rev(xs)@ys \end{aligned}$$

Sei $xs = x :: nil$. Dann

$$\begin{aligned} revi'(xs, ys) &= revi'(xr, x :: ys) && \text{Definition von } revi' \\ &= rev(xr)@(x :: ys) && \text{Induktionsannahme} \\ &= rev(xr)@([x]@ys) && \text{Definition von } @ \\ &= (rev(xr)@[x])@ys && @ \text{ assoziativ} \\ &= rev(x :: xr)@ys && \text{Definition von } rev \\ &= rev(xs)@ys \end{aligned}$$

□

Für längere Listen läuft die iterative Reversionsprozedur `revi` sehr viel schneller als die naive Reversionsprozedur `reverse` (siehe Abschnitt 7.6). Die vordeklarierte Reversionsprozedur `rev` ist in Moscow ML selbstverständlich iterativ realisiert.

Iterative Fibonacci-Prozedur

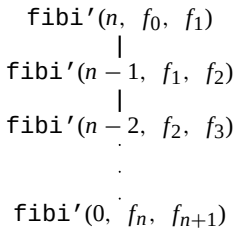
Wir entwickeln jetzt eine iterative Version der baumrekursiven Fibonacci-Prozedur:

```
fun fib n = if n<2 then n else fib(n-1) + fib(n-2)
```

Die dafür wichtige Idee sieht man am Besten mithilfe der mathematischen Definition der Fibonacci-Zahlen:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad \text{falls } n \geq 2 \end{aligned}$$

Die dritte Gleichung sagt uns, dass wir für $n \geq 2$ die nächste Fibonacci-Zahl aus den zwei vorhergehenden Fibonacci-Zahlen bestimmen können. Wir benötigen also eine Prozedur, die über zwei Akkumulatorargumente die jeweils vorhergehenden Fibonacci-Zahlen als Eingabe bekommt. Diese Idee wird durch den folgenden Rekursionsbaum dargestellt:



Wir realisieren unsere Idee wie folgt:

```
fun fibi'(n,a,b) = if n<1 then a else fibi'(n-1, b, a+b)

fun fibi n = fibi'(n,0,1)
```

Offensichtlich hat `fibi` für Argumente $n \in \mathbb{N}$ die Laufzeit $\Theta(n)$. Erwartungsgemäß berechnet Moscow ML

```
fibi 40
102334155 : int
```

sehr schnell. Dagegen benötigt die Berechnung von `fib 40` einige Minuten. Man kann zeigen, dass `fib` exponentielle Laufzeit hat.

Um die Korrektheit von `fibi` zu zeigen, beweisen wir die folgende Aussage:

$$\forall n, k \in \mathbb{N}\colon\ \mathit{fibi}'(n,\ f_k,\ f_{k+1}) = f_{n+k} \tag{7.5}$$

Beweis Durch Induktion über n.

Sei $n = 0$. Dann $\mathit{fibi}'(n,\ f_k,\ f_{k+1}) = f_k = f_{n+k}$ mithilfe der Definition von fibi'.

Sei $n > 0$. Dann

$$\begin{aligned}
\mathit{fibi}'(n,\ f_k,\ f_{k+1}) &= \mathit{fibi}'(n-1,\ f_{k+1},\ f_{k+1} + f_k) && \text{Definition } \mathit{fibi}' \\
&= \mathit{fibi}'(n-1,\ f_{k+1},\ f_{k+2}) \\
&= f_{n-1+k+1} && \text{Induktionsannahme} \\
&= f_{n+k}
\end{aligned}$$

□

7.13 Höherstufige Prozeduren

Bisher haben wir keine Beispiele mit höherstufigen Prozeduren betrachtet. Wir holen das jetzt mit der iterativen Faltungsprozedur

```
fun foldl f s nil     = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

nach. Zunächst kümmern wir uns um die Laufzeit. Sei

p: t_1 `*` t_2 `->` t_2

eine Prozedur, die für alle Argumente die Laufzeit 0 hat. Dann wertet ein Ausdruck

`foldl` *p s*

offensichtlich zu einer Prozedur vom Typ t_1 `list ->` t_2 aus, die für alle Argumente *xs* die Laufzeit $|xs| + 1$ hat. Wenn p eine Prozedur mit konstanter Laufzeit ist, die für alle Argumente terminiert, gilt allgemeiner, dass die Prozedur

`foldl` *p s*

für alle Argumente *xs* terminiert und die Laufzeit $\Theta(|xs|)$ hat.

In Abschnitt 4.5 haben wir gelernt, dass die Prozedur

```
foldl op:: nil
```

Listen reversiert. Wir wissen jetzt, dass sie dies mit linearer Laufzeit tut, also viel schneller ist als die naive Reversionsprozedur in Abschnitt 7.6, die quadratische Laufzeit hat. Das erklärt die Beobachtungen in Abschnitt 4.10. Die iterative Reversionsprozedur `revi'` in Abschnitt 7.12 kann man als eine expandierte Variante der Prozedur `foldl op:: nil` auffassen.

Wir wollen jetzt beweisen, dass die Prozedur `foldl op:: nil` die Funktion *rev* (siehe Abschnitt 7.5) berechnet. Um den Beweis auf eine solide Grundlage zu stellen, definieren wir die den monomorphen Instanzen der Prozeduren `foldl` und `op::` zugeordneten Funktionen.

Sei X eine Menge. Wir definieren:

$$\begin{aligned}
&\mathit{foldl} \in (X \times \mathcal{L}(X) \rightharpoonup \mathcal{L}(X)) \rightharpoonup \mathcal{L}(X) \rightharpoonup \mathcal{L}(X) \rightharpoonup \mathcal{L}(X)\\
&\quad \mathit{foldl}\ f\ ys\ \mathit{nil} = ys\\
&\mathit{foldl}\ f\ ys\ (x :: xr) = \mathit{foldl}\ f\ (f(x, ys))\ xr
\end{aligned}$$

$$\begin{aligned}
&\mathit{cons} \in X \times \mathcal{L}(X) \to \mathcal{L}(X)\\
&\mathit{cons}(x, xr) = x :: xr
\end{aligned}$$

Proposition 7.13.1 $\forall xs, ys \in \mathcal{L}(X)\colon \quad \mathit{foldl\ cons\ ys\ xs} = (\mathit{rev\ xs})@ys.$

Beweis Durch Induktion über xs.

Sei $xs = nil$. Dann

$$\begin{aligned}
\mathit{foldl\ cons\ ys\ xs} &= ys && \text{Definition von } \mathit{foldl}\\
&= \mathit{nil}@ys && \text{Definition von } @\\
&= (\mathit{rev\ xs})@ys && \text{Definition von } \mathit{rev}
\end{aligned}$$

Sei $xs = x :: xr$. Dann

$$\begin{aligned}
\mathit{foldl\ cons\ ys\ xs} &= \mathit{foldl\ cons}\ (x :: ys)\ xr && \text{Definition von } \mathit{foldl} \text{ und } \mathit{cons}\\
&= (\mathit{rev\ xr})@(x :: ys) && \text{Induktionsannahme}\\
&= (\mathit{rev\ xr})@([x]@ys) && \text{Definition von } @\\
&= ((\mathit{rev\ xr})@[x])@ys && \text{Assoziativität von } @\\
&= (\mathit{rev\ xs})@ys && \text{Definition von } \mathit{rev}
\end{aligned}$$

□

Proposition 7.13.2 $\forall xs \in \mathcal{L}(X)\colon \quad \mathit{foldl\ cons\ nil\ xs} = \mathit{rev\ xs}.$

Beweis

$$\begin{aligned}
\mathit{foldl\ cons\ nil\ xs} &= (\mathit{rev\ xs})@\mathit{nil} && \text{Proposition 7.13.1}\\
&= \mathit{rev\ xs} && \text{Proposition 7.5.1 (2)}
\end{aligned}$$

□

7.14 Laufzeiten der Sortierprozeduren

In Abschnitt 4.10 haben wir beobachtet, dass Sortieren durch Mischen für bestimmte Listen viel schneller ist als Sortieren durch Einfügen. Wir können jetzt erklären, warum das so ist.

Sortieren durch Einfügen

Zuerst betrachten wir die Hilfsprozedur

```
fun insert (x, nil)   = [x]
  | insert (x, y::yr) = if x<=y then x::y::yr
                        else y::insert(x,yr)
val insert : int * int list -> int list
```

Für eine aufsteigend sortierte Liste $xs \in \mathcal{L}(\mathbb{Z})$ und eine Zahl x, die größer als alle Elemente von xs ist, hat `insert`(x, xs) die Laufzeit $|xs|+1$. Für eine absteigend sortierte Liste $xs \in \mathcal{L}(\mathbb{Z})$ und eine Zahl x, die kleiner gleich als alle Elemente von xs ist, hat `insert`(x, xs) dagegen die Laufzeit 1. Die Prozedur `insert` hat also für Argumente (x, xs) die Laufzeit $O(|xs|)$, und dabei handelt es sich um eine scharfe obere Schranke.

Jetzt betrachten wir die Prozedur `isort`:

```
fun isort nil     = nil
  | isort (x::xr) = insert(x, isort xr)
val isort : int list -> int list
```

Der Rekursionsbaum von `isort`(xs) ist linear und hat die Größe $|xs|+1$. Als Beispiel betrachten wir

```
isort [3,2,1]
      |
 isort [2,1]
       |
  isort [1]
        |
   isort []
```

Da die Laufzeit von `insert` von der Anordnung der zu sortierenden Liste abhängen, hängt auch die Laufzeit von `isort` von der Anordnung ab. Wir betrachten den Rekursionsbaum von `isort` *xs* für den Fall, dass *xs* wie im Beispiel eine absteigend sortierte Liste ist. Dann muss `insert` für jeden Aufruf des Rekursionsbaums ein Element in eine aufsteigend sortierte Liste einfügen, das größer ist als jedes Listenelement. Jeder Aufruf `isort` *ys* im Rekursionsbaum verursacht daher die *Nebenkosten* $|ys|$ für den Aufruf von `insert`. Da die Prozedur `isort` ihre Argumentliste bei jedem rekursiven Aufruf um ein Element verkürzt, fallen im Rekursionsbaum für `isort` *xs* insgesamt die folgenden Nebenkosten für `insert` an:

$$|xs| + (|xs|-1) + (|xs|-2) + \cdots + 1 \;=\; \sum_{i=1}^{|xs|} i \;=\; \frac{1}{2} \cdot |xs| \cdot (|xs|-1)$$

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys))
                     (nil, nil) xs
val split : 'a list -> 'a list * 'a list

fun merge (nil , ys ) = ys
  | merge (xs , nil ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)
val merge : int list * int list -> int list

fun msort nil = nil
  | msort (x::nil) = x::nil
  | msort xs = let
                         val (ys,zs) = split xs
                       in
                         merge(msort ys, msort zs)
                       end
val msort : int list -> int list
```

Abbildung 7.2: Sortieren durch Mischen

ieren durch ügen hat nstens dratische fzeit (scharf)

Also hat `isort` für absteigend sortierte Listen *xs* die Laufzeit $\Theta(|xs|^2)$. Für nicht absteigend sortierte Listen hat `isort` eine kleinere Laufzeit, da die Kosten für `insert` kleiner sind. Also ist $O(|xs|^2)$ eine scharfe obere Schranke für die Laufzeit von `isort`. Für aufsteigend sortierte Listen *xs* hat `isort` die Laufzeit $\Theta(|xs|)$. Dies ist der günstigste Fall.

Sortieren durch Mischen

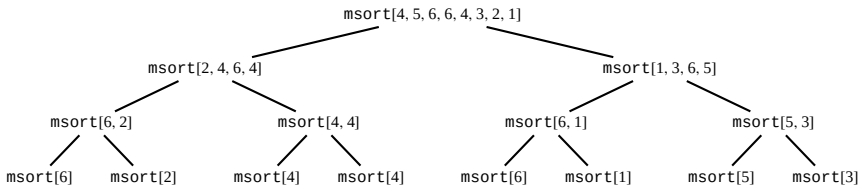
Wir zeigen jetzt, dass Sortieren durch Mischen (siehe Abbildung 7.2 und Abschnitt 4.7) die Laufzeit $\Theta(n \log n)$ hat, wobei n die Länge der zu sortierenden Listen ist.

Die folgenden Fakten lassen sich leicht verifizieren:

1. `split` *xs* hat die Laufzeit $|xs| + 1$.

2. Die Laufzeit von `merge`(xs, ys) hängt von der Anordnung der Listen *xs* und *ys* ab. Sie ist mindestens $\min\{|xs|, |ys|\} + 1$ und höchstens $|xs| + |ys| + 1$.

Sei $k \in \mathbb{N}$ und sei $xs \in \mathcal{L}(\mathbb{Z})$ mit $|xs| = 2^k$. Wir bestimmen die maximale Laufzeit von `msort` *xs*. Zur Einstimmung betrachten wir den Rekursionsbaum für eine Liste der Länge 2^3:



Der Rekursionsbaum für `msort` *xs* ist offensichtlich ein balancierter Binärbaum der Tiefe k. Seine Größe ist $2^{k+1} - 1$ (Proposition 5.4.4). Wir unterteilen den Baum in $k+1$ *Ebenen* wie folgt: Die nullte Ebene besteht aus der Wurzel, die erste Ebene aus den Nachfolgern der Wurzel, und die k-te Ebene aus den Blättern. Die n-te Ebene enthält 2^n Aufrufe von `msort`, wobei die Argumentlisten jeweils die Länge 2^{k-n} haben. Für die k-te Ebene fallen keine Nebenkosten an. Für die n-te Ebene mit $n < k$ fallen insgesamt höchstens die folgenden Nebenkosten an:

$$2^n \cdot (\underbrace{2^{k-n} + 1}_{\text{split}} + \underbrace{2^{k-n-1} + 2^{k-n-1} + 1}_{\text{merge}}) \;=\; 2^{k+1} + 2^{n+1}$$

Damit fallen im Rekursionsbaum insgesamt die folgenden Nebenkosten an:

$$\sum_{n=0}^{k-1}(2^{k+1} + 2^{n+1}) \;=\; k \cdot 2^{k+1} + \sum_{n=0}^{k-1} 2^n \;=\; k \cdot 2^{k+1} + 2^k - 1$$

($\sum_{n=0}^{k-1} 2^n = 2^k - 1$, Beweis durch Induktion über n) Damit hat `msort` *xs* für Listen der Länge 2^k höchstens die Laufzeit

$$2^{k+1} - 1 + k \cdot 2^{k+1} + 2^k - 1 \;=\; 2k \cdot 2^k + 3 \cdot 2^k - 2$$

Mit einer analogen Argumentation kann man zeigen, dass `msort` *xs* für Listen der Länge 2^k mindestens die folgenden Laufzeit hat:

$$\frac{3}{2} \cdot k \cdot 2^k + 4 \cdot 2^k - 3$$

Wir definieren

$$\begin{aligned} l &= \lambda k \in \mathbb{N}.\ \frac{3}{2} \cdot k \cdot 2^k + 4 \cdot 2^k - 3 \\ u &= \lambda k \in \mathbb{N}.\ 2k \cdot 2^k + 3 \cdot 2^k - 2 \end{aligned}$$

Sei $\phi_1(n)$ die minimale und $\phi_2(n)$ die maximale Laufzeit von `msort` für Listen der Länge $n \in \mathbb{N}$. Sowohl ϕ_1 als auch ϕ_2 sind monoton wachsend. Damit haben wir:

$$\forall n \geq 2\colon\ l(\lfloor \log_2 n \rfloor) \leq \phi_1(n) \leq \phi_2(n) \leq u(\lceil \log_2 n \rceil)$$

Da ($\log_2 n$ steht im Folgenden für if $n = 0$ then 0 else $\log_2 n$)

$$\Theta(l(\lfloor \log_2 n \rfloor)) = \Theta(u(\lceil \log_2 n \rceil)) = \Theta(n \log n)$$

gilt $\Theta(\phi_1) = \Theta(\phi_2) = \Theta(n \log n)$. Damit hat `msort` für Listen `xs` die Laufzeit $\Theta(|xs| \cdot \log |xs|)$.

7.15 Zusammenfassung

In diesem Kapitel haben wir zwei wichtige Aspekte von Prozeduren untersucht: Korrektheit und Laufzeit. Wir haben nur funktionale Prozeduren betrachtet, also Prozeduren, die als prozedurale Interpretationen von Funktionsdefinitionen erhalten werden. Für die in diesem Kapitel untersuchten Fragen ist es unerheblich, ob Prozeduren in einer konkreten Programmiersprache oder mithilfe von mathematischer Notation definiert sind.

Für Fragen der Korrektheit genügt die funktionale Interpretation von Prozedurdeklarationen. Es geht also darum, Eigenschaften von Funktionen zu beweisen. Für den Beweis von Eigenschaften von rekursiv definierten Funktionen benötigt man Induktion. Für Funktionen, die auf Listen definiert sind, haben wir Listeninduktion verwendet.

Manchmal kann man die Induktion einsparen, indem man zunächst zeigt, dass die zugeordneten Prozeduren für bestimmte Argumente terminieren. Terminierungsbeweise haben wir mithilfe von wohlfundierten Relationen geführt.

Laufzeiteigenschaften beziehen sich im Gegensatz zu Korrektheitseigenschaften auf die prozedurale Interpretation von Funktionsdefinitionen.

In der Praxis haben die meisten Prozeduren lineare Laufzeit. Prozeduren mit logarithmischer Laufzeit sind besonders schnell. Logarithmische Laufzeit kommt wie bei der schnellen Exponentiation meistens dadurch zustande, dass bei jedem Rekursionsschritt die Größe des Arguments halbiert wird. Die Laufzeit $\Theta(n \log n)$ ist in der Praxis kaum von linearer Laufzeit unterscheidbar. Ein typisches Beispiel ist Sortieren durch Mischen. Quadratische Laufzeit ist in der Praxis oft schon zu langsam (siehe Abschnitt 4.10). Exponentielle Laufzeit ist fast immer zu langsam. Wenn man eine Prozedur schreibt, sollte man grundsätzlich auf ihre asymptotische Laufzeit achten. Prozeduren mit quadratischer oder schlechterer Laufzeit sind Kandidaten für Performanzprobleme.

Oft ist es möglich, eine langsame Prozedur durch eine schnellere Prozedur zu ersetzen. Wir haben dies an einigen Beispielen gesehen: Reversieren von Listen

(lineare statt quadratische Laufzeit), Sortieren von Listen ($\Theta(n \log n)$ statt quadratische Laufzeit), Fibonacci-Zahlen (lineare statt exponentielle Laufzeit) und Exponentiation (logarithmische statt lineare Laufzeit).

Einige interessante und in der Praxis relevante Programmierprobleme können allerdings nur mit exponentieller Laufzeit gelöst werden.

Wenn mit größerer Rekursionstiefe gerechnet werden soll, sollte man unbedingt mit iterativer Rekursion arbeiten. Der Übergang von linearer zu iterativer Rekursion gelingt oft mithilfe von Akkumulatorargumenten.

Die Konstruktion und Analyse von Algorithmen ist ein wichtiges und reiches Teilgebiet der Informatik. Wir haben hier nur die Spitze der Spitze des Eisbergs gesehen. Wenn Sie mehr sehen wollen, empfehlen wir Ihnen das Buch von Cormen, Leiserson, und Rivest [**?**] sowie das Buch von Okasaki [**?**].

Übungsaufgaben

Aufgabe 7.1 (Rekursionsbäume und Laufzeit) Sei die folgende Prozedur gegeben:

```
fun p n = if n< 4 then () else (p(n-1) ; p(n div 2))
```

1. Zeichnen Sie den Rekursionsbaum für den Aufruf `p 8`.
2. Geben Sie die Laufzeit des Aufrufs `p 8` an.
3. Schreiben Sie eine Prozedur `phi: int->int`, die die Laufzeit von `p` berechnet.

Aufgabe 7.2 (Rekursionsbäume und Laufzeit) Sei die folgende Prozedur gegeben:

```
fun f(m,n) = if m=0 then n
             else if n=0 then m
                  else if m<n then f(m-1,n+1)
                       else f(m+1,n-1)
```

1. Zeichnen Sie den Rekursionsbaum für `f(3,45)`.
2. Ist `f` iterativ?
3. Geben Sie die exakte Laufzeit von `f` für Argumente $(x, y) \in \mathbb{N}^2$ an.
4. Geben Sie eine Prozedur `g` an, die für alle Argumente $(x, y) \in \mathbb{N}^2$ dasselbe Ergebnis wie `f` berechnet, aber mit konstanter Laufzeit.

Aufgabe 7.3 (Iteratives Tabulate) Sie sollen eine Prozedur

```
tabulate : (int -> 'a) -> int -> 'a list
```

schreiben, die nur iterative Rekursion benutzt und für $n \in \mathbb{N}$ die folgende Gleichung erfüllt:

$$\texttt{tabulate}\ f\ n = [f\ 0, \ldots, f\ n]$$

Aufgabe 7.4 (Reversierendes Map) Sie sollen eine Prozedur

```
rmap : ('a -> 'b) -> 'a list -> 'b list
```

schreiben, die nur iterative Rekursion benutzt und die folgende Gleichung erfüllt:

$$\texttt{rmap}\ f\ [x_1, \ldots, x_n] = [f\ x_n, \ldots, f\ x_1]$$

1. Schreiben Sie `rmap` mit `foldl`.
2. Schreiben Sie `rmap` mithilfe einer iterativen Hilfsprozedur.

Aufgabe 7.5 (Iteratives Berechnen von balancierten Binärbäumen) Gegeben sei der Typkonstruktor `tree` für Binärbäume aus Abschnitt 6.5.

1. Schreiben Sie eine Prozedur

   ```
   tree : int -> unit tree
   ```

   die für $n \in \mathbb{N}$ einen balancierten Binärbaum der Tiefe n mit linearer Laufzeit berechnet. Dabei soll nur iterative Rekursion verwendet werden. Verwenden Sie eine Hilfsprozedur `tree'` mit einem Akkumulatorargument.

2. Schreiben Sie eine Prozedur

   ```
   tree : int -> int tree
   ```

   die für $n \in \mathbb{N}$ einen balancierten Binärbaum der Tiefe n berechnet, dessen Knoten mit der Tiefe des jeweils erreichbaren Teilbaums markiert sind. Beispielsweise soll

   ```
   tree 2 = N(2, N(1, L 0, L 0), N(1, L 0, L 0))
   ```

   gelten. Dabei soll nur iterative Rekursion verwendet werden und die Laufzeit soll linear sein. Verwenden Sie eine Hilfsprozedur `tree'` mit zwei Akkumulatorargumenten.

Aufgabe 7.6 (Induktion über Listen) Beweisen Sie die Teile (2)-(6) von Proposition 7.5.1. Dabei dürfen Sie Teil (1) der Proposition verwenden.

Aufgabe 7.7 (Iteratives Length) Sei X eine Menge.

1. Definieren Sie mit iterativer Rekursion eine Funktion

   $$len' \in \mathcal{L}(X) \times \mathbb{Z} \to \mathbb{Z}$$

   für die gilt:

   $$\forall xs \in \mathcal{L}(X)\ \forall n \in \mathbb{Z}\colon\ len'(xs, n) = |xs| + n$$

2. Beweisen Sie mit Listeninduktion, dass ihre Funktion diese Gleichung erfüllt.
3. Schreiben Sie eine Prozedur `len: 'a list -> int`, die die Länge eine Liste mithilfe einer iterativen Hilfsprozedur berechnet. Die Hilfsprozedur soll ihre Funktion len' berechnen. Geben Sie exakte Laufzeit ihrer Prozedur `len` an.

Aufgabe 7.8 (Größter gemeinsamer Teiler) Sei die folgende Prozedur gegeben:

```
fun gcd(x,y) = if x=y then x
               else if x<y then gcd(x, y-x) else gcd(x-y, y)
```

1. Ist `gcd` iterativ?
2. Zeichnen Sie den Rekursionsbaum für den Aufruf `gcd(216,60)`.
3. Schreiben Sie eine iterative Prozedur `phi: int*int->int`, die die Laufzeit von `gcd` berechnet.
4. Zeigen Sie, dass `gcd` für alle Argumente $(x, y) \in \mathbb{N}^+ \times \mathbb{N}^+$ terminiert. Geben Sie dazu die Terminierungsbedingungen und eine passende wohlfundierte Relation an.
5. Sei ϕ die Laufzeit von `gcd`. Beweisen Sie durch Induktion über $\max\{x, y\}$:

   $$\forall x, y \in \mathbb{N}^+\colon\ \phi(x, y) \leq \max\{x, y\}$$

6. Geben Sie für alle $x \in \mathbb{N}^+$ ein $y \in \mathbb{N}^+$ an, sodass $\phi(x, y) = \max\{x, y\}$.
7. Beweisen Sie, dass `gcd` für alle Argumente $(x, y) \in \mathbb{N}^+ \times \mathbb{N}^+$ den größten gemeinsamen Teiler von x und y berechnet. Nehmen Sie dazu an, dass `gcd` für diese Argumente terminiert (siehe Teil (4)).

Aufgabe 7.9 (Laufzeitsynthese)

1. Schreiben Sie möglichst einfache Prozeduren `int -> unit`, die für $n \in \mathbb{N}$ die folgenden Laufzeiten haben:

$\Theta(\log n)$	`fun tlogn n =` $\cdots$
$\Theta(n)$	`fun tn n =` $\cdots$
$\Theta(n \log n)$	`fun tnlogn n =` $\cdots$
$\Theta(n^2)$	`fun tn2 n =` $\cdots$
$\Theta(n^3)$	`fun tn3 n =` $\cdots$
$\Theta(2^n)$	`fun t2n n =` $\cdots$
$\Theta(3^n)$	`fun t3n n =` $\cdots$

   Sie können Hilfsprozeduren verwenden. Die folgende Fakten sind hilfreich:

   a) $\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1)$ für $n > 0$.

   b) Balancierte Binärbäume der Tiefe n haben die Größe $2^{n+1} - 1$.

   c) Balancierte Ternärbäume der Tiefe n haben die Größe $\frac{1}{2}(3^{n+1} - 1)$.

2. Ermitteln Sie für alle Prozeduren außer `tlogn` Argumente, für die diese eine beobachtbare Laufzeit zwischen 2 und 60 Sekunden haben.

3. Schreiben Sie ihre Prozeduren zu Prozeduren des Typs `int -> int` um, die die jeweils geforderte Laufzeit haben und zusätzlich die Laufzeit des Aufrufs liefern.