10 Eigenschaften von Prozeduren

Daß $1+1$ gleich 2 ist, ist ein *Beispiel* für die Arbeitsweise der Addition. Dieses Beispiel könnte auch als Testfall für eine programmierte Version der Addition durchgehen. Unter Umständen kann ein Beispiel, als Testfall formuliert, einen Fehler in einem Programm finden. Allerdings ist das Formulieren von Beispielen mühsam. Schlimmer noch, eine Menge von Testfällen reicht nur selten aus, um die Korrektheit einer Prozedur auch zu garantieren: Die Testfälle decken meist nicht alle möglichen Anwendungen einer Prozedur ab. Darum ist es oft sinnvoll, statt isolierter Beispiele allgemeine Eigenschaften zu formulieren und zu überprüfen – am besten sogar, diese zu beweisen. Dieses Kapitel zeigt, wie das geht.

10.1 Eigenschaften von eingebauten Operationen

In diesem Abschnitt wird die Formulierung und Überprüfung von Eigenschaften anhand von bekannten eingebauten Operationen wie `+`, `and`, `=` etc. demonstriert.

10.1.1 Binäre Operationen

Eine allgemein bekannte Eigenschaft der Addition ist die *Kommutativität*:

$$a+b=b+a$$

Auch wenn intuitiv die Bedeutung klar ist, ist die Eigenschaft genau genommen so noch nicht präzise schriftlich festgehalten, da nicht notiert ist, was a und b sind: Die Idee ist natürlich, daß a und b beliebige *Zahlen* sind. Im allgemeinen also:

$$(\forall a \in \mathbb{C}, b \in \mathbb{C})\ a+b=b+a$$

(Wer sich mit der Vorstellung komplexer Zahlen nicht wohlfühlt, kann das $\mathbb{C}$ auch durch $\mathbb{R}$ oder $\mathbb{Q}$ ersetzen.)

In Scheme läßt sich diese Eigenschaft für die eingebaute Prozedur `+` aufschreiben – das $\forall$ ist auf den meisten Tastaturen nicht vertreten und wird darum ausgeschrieben (siehe Abbildung 10.1):

```
(for-all ((a number)
          (b number))
  (= (+ a b) (+ b a)))
```

Das Ergebnis dieses Ausdrucks wird in der REPL etwas undurchsichtig angezeigt:

`For-all` ermöglicht das Formulieren von *Eigenschaften*. Ein `for-all`-Ausdruck hat die folgende allgemeine Form:

`(for-all ((`v_1 c_1`) ... (`v_n c_n`)) `b`)`

Dabei müssen die v_i Variablen sein, die c_i Verträge und b (der Rumpf) ein Ausdruck, der entweder einen booleschen Wert oder eine Eigenschaft liefert. Der `for-all`-Ausdruck hat als Wert eine Eigenschaft, die besagt, daß b gilt für *alle* Werte der v_i, welche die Verträge c_i erfüllen.

Abbildung 10.1 `for-all`

`Check-property` testet eine Eigenschaft analog zu `check-expect`. Eine `check-property`-Form sieht so aus:

`(check-property `e`)`

e ist ein Ausdruck, der eine Eigenschaft liefern muß – in der Regel also ein `for-all`-Ausdruck.
Bei der Auswertung setzt `check-property` für die Variablen der `for-all`-Ausdrücke verschiedene Werte ein und testet, ob die Eigenschaft jeweils erfüllt ist.
`Check-property` funktioniert nur für Eigenschaften, bei denen aus den Verträgen sinnvoll Werte generiert werden können. Dies ist für die meisten eingebauten Verträge der Fall, aber nicht für Vertragsvariablen und Verträge, die mit `predicate`, `property` oder `define-record-procedures` definiert wurden.

Abbildung 10.2 `check-property`

```
↪ #<:property>
```

Diese Eigenschaft läßt sich mit Hilfe der `check-property`-Form (siehe Abbildung 10.2) überprüfen:

```
(check-property
  (for-all ((a number)
            (b number))
    (= (+ a b) (+ b a))))
```

`Check-property` fungiert, wie `check-expect` oder `check-within`, als Testfall und wird auch als solcher ausgewertet. Da `+` tatsächlich kommutativ ist, läuft der Testfall auch anstandslos durch.

Interessanter wird es erst bei Eigenschaften, die nicht stimmen. Zum Beispiel ist die Subtraktion `-` nicht kommutativ:

```
(check-property
 (for-all ((a number)
           (b number))
   (= (- a b)
      (- b a))))
```

Hierfür liefert DrScheme folgende Meldung:

```
Eigenschaft falsifizierbar mit a = 0.0 b = -1.5
```

Falsifizierbar bedeutet, daß es ein *Gegenbeispiel* für die Eigenschaft gibt, also Werte für die Variablen `a` und `b`, welche die Eigenschaft falsch werden lassen. DrScheme hat in diesem Fall ein Gegenbeispiel gefunden, bei dem `a` den Wert 0.0 und `b` den Wert 1.5 hat:

```
(- 0.0 -1.5)
↪ 1.5
(- -1.5 0.0)
↪ -1.5
```

Dieses Beispiel widerlegt also tatsächlich die Behauptung der Eigenschaft.

Hinter den Kulissen hat DrScheme verschiedene Werte für `a` und `b` durchprobiert und in die Eigenschaft eingesetzt, also effektiv nach einem Gegenbeispiel gesucht. Für die Kommutativität von `+` gibt es kein Gegenbeispiel – DrScheme konnte also auch keins finden. Daß ausgerechnet das merkwürdige Beispiel 0.0 und -1.5 herauskam, liegt an der relativ komplexen Suchstrategie von DrScheme.

Auf diese Art und Weise lassen sich eine Reihe von interessanten Eigenschaften formulieren, so zum Beispiel die Assoziativität von `+`:

```
(check-property
 (for-all ((a number)
           (b number)
           (c number))
    (= (+ a (+ b c))
       (+ (+ a b) c))))
```

Hierbei gibt es allerdings eine böse Überraschung – DrScheme produziert ein Gegenbeispiel:

```
Eigenschaft falsifizierbar mit
  a = 2.6666666666666665 b = 6.857142857142857 c = -6.857142857142857
```

Es ist kein Zufall, daß es sich um Zahlen mit vielen Nachkommastellen handelt. Wenn dieses Gegenbeispiel in die Eigenschaft eingesetzt wird, liefert die REPL folgende Ergebnisse:

```
(+ 2.6666666666666665 (+ 6.857142857142857 -6.857142857142857))
↪ 2.6666666666666665
(+ (+ 2.6666666666666665 6.857142857142857) -6.857142857142857)
↪ 2.666666666666667
```

Hier wird sichtbar daß, wie bereits in Abschnitt 2.2 erwähnt, bei Berechnungen mit sogenannten *inexakten Zahlen*, das sind Zahlen mit einem Dezimalpunkt, die mathematischen Operationen nur mit einer begrenzten Anzahl von Stellen durchgeführt werden und dann

runden – da auch noch binär und nicht dezimal gerundet wird, sieht das Ergebnis dieser Rundung oft unintuitiv aus. Dieses Beispiel zeigt nun, daß Addition plus binäre Rundung nicht assoziativ ist. Die Assoziativität gilt nur für das Rechnen mit *exakten Zahlen*. Immerhin sind alle Zahlen, die den Vertrag `rational` erfüllen, exakt, die Eigenschaft läßt sich also reformulieren:

```
(check-property
 (for-all ((a rational)
           (b rational)
           (c rational))
    (= (+ a (+ b c))
       (+ (+ a b) c))))
```

Und tatsächlich, in dieser Form wird die Eigenschaft nicht beanstandet.

Kommutativität und Assoziativität sind jeweils Eigenschaften einer einzelnen Operation, in diesem Fall `+`. Manche Eigenschaften beschreiben auch das Zusammenspiel mehrerer Operationen, wie zum Beispiel die Distributivität, die für Addition und Multiplikation gilt:

$$(\forall a \in \mathbb{C}, b \in \mathbb{C}, c \in \mathbb{C})\ a \cdot (b + c) = a \cdot b + b \cdot c$$

Auch dies läßt sich direkt nach Scheme übersetzen, diesmal gleich mit `rational` statt `number`:

```
(check-property
 (for-all ((a rational)
           (b rational)
           (c rational))
   (= (* a (+ b c))
      (+ (* a b) (* a c)))))
```

Auch hier hat DrScheme nichts zu meckern.

Diese drei Eigenschaften – Kommutativität, Assoziativität und Distributivität – tauchen immer wieder auf, da sie nicht nur für arithmetische Operationen gelten (auch die Multiplikation ist kommutativ und assoziativ) sondern auch anderswo. Zum Beispiel gelten Kommutativität und Assoziativität auch für das logische `and`:

```
(check-property
 (for-all ((a boolean)
           (b boolean))
    (boolean=? (and a b)
               (and b a))))

(check-property
 (for-all ((a boolean)
           (b boolean)
           (c boolean))
    (boolean=? (and a (and b c))
               (and (and a b) c))))
```

Hier muß die eingebaute Prozedur `boolean=?` verwendet werden, die boolesche Werte vergleicht, analog zu `=`, die nur Zahlen vergleichen kann.

Genau wie `+` und `*` erfüllen auch `and` und `or` die Distributivität:

```
(check-property
 (for-all ((a boolean)
           (b boolean)
           (c boolean))
   (boolean=? (and a (or b c))
              (or (and a b) (and a c)))))
```

Analoge Eigenschaften gelten für `or`.

Auch das DeMorgan'sche Gesetz (siehe Abschnitt B.1) läßt sich in Scheme formulieren:

```
(check-property
 (for-all ((a boolean)
           (b boolean))
   (boolean=? (not (and a b))
              (or (not a) (not b)))))
```

Bei vielen Operationen ist außerdem interessant, ob sie ein *neutrales Element* besitzen, also ein Argument, das dafür sorgt, daß die Operation ein anderes Argument unverändert zurückgibt. Die Addition hat z.B. die 0 als neutrales Element:

```
(check-property
  (for-all ((a rational))
    (= (+ a 0) a)))
```

Streng genommen ist damit nur gesichert, daß 0 *rechtsneutrales Element* ist, also von rechts addiert das andere Argument unverändert herauskommt. Aus der Kommutativität folgt aber, daß jedes rechtsneutrale Element auch ein linksneutrales Element ist.

Bei manchen Operationen gibt es neben dem neutralen Element zu jedem Element auch ein *inverses Element*: Wenn eine binäre Operation auf ein Element und sein Inverses angewendet wird, so muß das neutrale Element herauskommen. Bei der Addition entsteht das Inverse zu einer Zahl durch Umdrehen des Vorzeichens:

```
(check-property
 (for-all ((a rational))
   (= (+ a (- a)) 0)))

(check-property
 (for-all ((a rational))
   (= (+ (- a) a) 0)))
```

Hier noch einmal eine Zusammenfassunge der in diesem Abschnitt behandelten Eigenschaften, mit Kurzfassungen der mathematischen Formulierungen:

Eine *Implikation* in einer Eigenschaft wird folgendermaßen geschrieben:

`(==>` e e_p`)`

Dabei muß e ein Ausdruck mit booleschem Wert sein (die *Voraussetzung*) und e_p eine Eigenschaft oder ein boolescher Ausdruck. Die Implikation liefert ihrerseits wieder eine Eigenschaft, die gilt, wenn e_p immer dann gilt, wenn die Voraussetzung erfüllt ist, also `#t` liefert.

Abbildung 10.3 `==>`

Mantra 11 (Eigenschaften von binären Operationen) Folgende Eigenschaften sind prinzipiell auf alle *binären* Operationen denkbar, die zwei Elemente einer Menge M akzeptieren und wiederum ein Element von M zurückgeben.

- Kommutativität $a \star b = b \star a$
- Assoziativität $(a \star b) \star c = a \star (b \star c)$
- Distributivität $a \otimes (b \star c) = (a \otimes b) \star (a \otimes c)$; $(b \star c) \otimes a = (b \otimes a) \star (c \otimes a)$
- neutrales Element ($a \star \nu = a$; $\nu \star a = a$)
- inverses Element $a \star a^{-1} = \nu$; $a^{-1} \star a = \nu$

10.1.2 Eigenschaften von binären Prädikaten

Die Prozedur `=` paßt nicht in das Scheme der Eigenschaften des folgenden Abschnitts. Sie hat folgenden Vertrag:

```
(: = (number number -> boolean))
```

Damit konsumiert sie zwar zwei Argumente aus derselben Menge, liefert aber einen booleschen Wert zurück. Stattdessen handelt es sich um ein *binäres Prädikat* bzw. eine *binäre Relation*. Für binäre Relationen kommt ein anderer Satz Eigenschaften in Frage. (Die mathematische Seite ist in Anhang B.5 beschrieben.) Insbesondere ist `=` eine *Äquivalenzrelation* und damit *reflexiv*, *symmetrisch* und *transitiv*.

Die Reflexivität besagt, daß jedes Element der Grundmenge (in diesem Fall die Menge der Zahlen) zu sich selbst in Beziehung steht:

```
(check-property
 (for-all ((a number))
   (= a a)))
```

Die Symmetrie bedeutet für `=`, daß aus `(= a b)` $\hookrightarrow$ `#t` das „Spiegelbild“ `(= b a)` $\hookrightarrow$ `#t` folgt. Mathematisch geschrieben sähe das so aus:

$$(\forall a \in \mathbb{C}, b \in \mathbb{C})\ a = b \Rightarrow b = a$$

Der Implikationspfeil $\Rightarrow$ wird in Scheme `==>` geschrieben. (Siehe Abbildung 10.3.) Der Test der Symmetrie sieht also folgendermaßen aus:

```
(check-property
 (for-all ((a number)
           (b number))
   (==> (= a b)
        (= b a))))
```

Ähnlich läuft es mit der Transitivität: Wenn zwei Zahlen *a* und *b* gleich sind sowie *b* und eine dritte Zahl *c*, dann müssen auch *a* und *c* gleich sein:

```
(check-property
 (for-all ((a number)
           (b number)
           (c number))
   (==> (and (= a b) (= b c))
        (= a c))))
```

Neben den drei Eigenschaften von Äquivalenzrelationen tritt auch gelegentlich die Eigenschaft *Antisymmetrie* auf (die mathematische Definition steht in Anhang B.5).

Mantra 12 (Eigenschaften von binären Prädikaten) Folgende Eigenschaften sind für binäre Prädikate denkbar:

- Reflexivität $a \leftrightsquigarrow a$
- Symmetrie $a \leftrightsquigarrow b \Rightarrow b \leftrightsquigarrow a$
- Transitivität $a \leftrightsquigarrow b \wedge b \leftrightsquigarrow c \Rightarrow a \leftrightsquigarrow c$
- Antisymmetrie $a \leftrightsquigarrow b \wedge b \leftrightsquigarrow a \Rightarrow a = b$

10.2 Eigenschaften von Prozeduren auf Listen

Es wird Zeit, Eigenschaften von selbstgeschriebenen Prozeduren zu überprüfen. In diesem Abschnitt geht es um einige der Prozeduren, die auf Listen operieren: `concatenate`, `invert`, und `list-sum`.

10.2.1 Eigenschaften von `concatenate`

Die Prozedur `concatenate` aus Abschnitt 7.2 hängt zwei Listen aneinander. Auch `concatenate` ist assoziativ: Wenn drei Listen mit Hilfe von `concatenate` aneinandergehängt werden, spielt es keine Rolle, ob zuerst die ersten beiden oder zuerst die letzten beiden Listen aneinandergehängt werden. Nach dem Muster der Assoziativität von `+` und `and` sieht der Test dieser Eigenschaft folgendermaßen aus:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number))
           (lis-3 (list number)))
```

```
   (... (concatenate (concatenate lis-1 lis-2) lis-3)
        (concatenate lis-1 (concatenate lis-2 lis-3)))))
```

Beim Test ist der Vertrag von `lis-1`, `lis-2` und `lis-3` jeweils `(list number)`. Der Vertrag von `concatenate`

```
(: concatenate ((list %a) (list %a) -> (list %a)))
```

suggeriert allerdings, daß der Vertrag von `lis-1`, `lis-2` und `lis-3` jeweils `(list %a)` lauten sollte, also allgemeiner als `(list number)`. Verträge mit Vertragsvariablen funktionieren allerdings nicht im Zusammenhang mit Eigenschaften, wie folgendes Beispiel zeigt:

```
(check-property
  (for-all ((x %a))
    ...))
```

Dieser Code liefert die Fehlermeldung „Vertrag hat keinen Generator“: Das liegt daran, daß die Vertragsvariable `%a` zuwenig Information über die zugrundeliegenden Werte liefert, als daß DrScheme sinnvoll Werte für die Tests generieren könnte. Aus diesem Grund müssen in `for-all` immer „konkrete“ Verträge ohne Vertragsvariablen angegeben werden. (Aus ähnlichen Gründen funktionieren auch einige andere Arten von Verträgen nicht bei `for-all`, inbesondere Record-Verträge. Prozedurverträge sind allerdings zulässig und werden in Abschnitt 10.3 behandelt.)

Für `concatenate` wäre es zwar gründlicher, die Tests auch noch für andere Sorten von Listenelementen als `number` durchzuführen – da aber `concatenate` mit den Listenelementen nichts anfängt, außer sie in weitere Liste zu stecken, reicht die Formulierung der Eigenschaft mit `(list number)` aus.

Es bleibt noch ein weiteres Problem bei der Formulierung der Assoziativität für `concatenate`: Es steht noch keine Prozedur für den Vergleich der beiden Listen zur Verfügung, die muß erst noch geschrieben werden. Kurzbeschreibung und Vertrag:

```
; Zwei Listen aus Zahlen vergleichen
(: number-list=? ((list number) (list number) -> boolean))
```

Die Testfälle sollten insbesondere Listen unterschiedlicher Länge berücksichtigen:

```
(check-expect (number-list=? empty empty) #t)
(check-expect (number-list=? (list 1.0 2.0 3.0) (list 1.0 2.0 3.0)) #t)
(check-expect (number-list=? (list 1.0 2.0 3.0) (list 1.0 2.0)) #f)
(check-expect (number-list=? (list 1.0 2.0) (list 1.0 2.0 3.0)) #f)
(check-expect (number-list=? (list 1.0 2.0 3.0) (list 1.0 2.1 3.0)) #f)
```

Die erste Schablone, ausgewählt nach dem ersten Listenparameter `lis-1`, sieht so aus:

```
(define number-list=?
  (lambda (lis-1 lis-2)
```

```
    (cond
      ((empty? lis-1)
       ...)
      ((pair? lis-1)
       ... (first lis-1) ...
       ... (number-list=? (rest lis-1) ...) ...))))
```

Die Schablone für den zweiten Listenparameter `lis-2` wird in beide Zweige des `cond` eingesetzt:

```
(define number-list=?
  (lambda (lis-1 lis-2)
    (cond
      ((empty? lis-1)
       (cond
         ((empty? lis-2) ...)
         ((pair? lis-2)
          ... (first lis-2) ...
          ... (number-list=? ... (rest lis-2)))))
      ((pair? lis-1)
       ... (first lis-1) ...
       ... (number-list=? (rest lis-1) ...) ...
       (cond
         ((empty? lis-2) ...)
         ((pair? lis-2)
          ... (first lis-2) ...
          ... (number-list=? ... (rest lis-2))))))))
```

Es gibt also insgesamt vier Fälle bei den Verzweigungen:

- Im ersten Fall sind beide Listen leer, das Ergebnis ist also `#t`.
- Im zweiten Fall ist die erste Liste leer und die zweite nichtleer. Das Ergebnis ist also `#f` und die Schablonenelemente sind überflüssig.
- Im dritten Fall ist die erste Liste nichtleer und die zweite leer. Das Ergebnis ist also wiederum `#f`.
- Im vierten Fall sind beide Listen nichtleer und in der Schablone stehen die jeweils ersten Elemente von `lis-1` und `lis-2`. Die beiden Listen sind nur gleich, wenn die beiden ersten Elemente gleich sind. Außerdem müssen natürlich die beiden Reste der Listen ebenfalls gleich sind – die beiden rekursiven Aufrufe aus den Schablonen können also kombiniert werden:

```
(define number-list=?
  (lambda (lis-1 lis-2)
    (cond
      ((empty? lis-1)
```

```
       (cond
         ((empty? lis-2) #t)
         ((pair? lis-2) #f)))
      ((pair? lis-1)
       (cond
         ((empty? lis-2) #f)
         ((pair? lis-2)
          (and (= (first lis-1) (first lis-2))
               (number-list=? (rest lis-1) (rest lis-2)))))))))
```

Damit kann jetzt die Assoziativität von `concatenate` getestet werden:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number))
           (lis-3 (list number)))
   (number-list=? (concatenate (concatenate lis-1 lis-2) lis-3)
                  (concatenate lis-1 (concatenate lis-2 lis-3)))))
```

`Concatenate` hat außerdem ein neutrales Element, und zwar sowohl im linken als auch im rechten Argument:

```
(check-property
 (for-all ((lis (list number)))
   (number-list=? lis (concatenate empty lis))))

(check-property
 (for-all ((lis (list number)))
   (number-list=? lis (concatenate lis empty))))
```

`Concatenate` ist allerdings demonstrierbar nicht kommutativ. Der entsprechende Test sieht so aus:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number)))
   (number-list=? (concatenate lis-1 lis-2)
                  (concatenate lis-2 lis-1))))
```

DrScheme liefert hierfür ein Gegenbeispiel:

```
Eigenschaft falsifizierbar mit
         lis-1 = #<list -3.75> lis-2 = #<list 1.5 1.5>
```

10.2.2 Eigenschaften von `number-list=?`

Wie der Zufall so will, hat auch die Hilfsprozedur `number-list=?` interessante Eigenschaften: Wie `=` muß auch `number-list=?` eine Äquivalenzrelation sein – schließlich te-

stet sie wie = auf Gleichheit. Die dazugehörigen Eigenschaften – Reflexivität, Symmetrie und Transitivität – können ebenso wie bei = formuliert werden:

Reflexivität:

```
(check-property
 (for-all ((lis (list number)))
   (number-list=? lis lis)))
```

Symmetrie:

```
(check-property
  (for-all ((lis-1 (list number))
            (lis-2 (list number)))
    (==> (number-list=? lis-1 lis-2)
         (number-list=? lis-2 lis-1))))
```

Transitivität

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number))
           (lis-3 (list number)))
   (==> (and (number-list=? lis-1 lis-2)
             (number-list=? lis-2 lis-3))
        (number-list=? lis-1 lis-3))))
```

10.2.3 Eigenschaften von `invert`

Die Prozedur `invert` aus Abschnitt 8.1 dreht die Reihenfolge der Elemente einer Liste um. Eine naheliegende Eigenschaft von `invert` ist, daß zweimaliges Umdrehen wieder die Ursprungsliste liefern sollte:

```
(check-property
 (for-all ((lis (list number)))
   (number-list=? lis (invert (invert lis)))))
```

Auch bei `invert` enthält der Vertrag eine Vertragsvariable:

```
(: invert ((list %a) -> (list %a)))
```

Genau wie bei `concatenate` macht `invert` mit den Listenelementen nichts spezielles, es können also auch zum Beispiel Zeichenketten benutzt werden. Diese Änderung allein funktioniert allerdings nicht:

```
(check-property
 (for-all ((lis (list string)))
   (number-list=? lis (invert (invert lis)))))
```

`Expect` liefert eine Eigenschaft analog zur Funktionsweise von `check-expect`. Ein `expect`-Ausdruck hat folgende Form:

`(expect` e_1 e_2`)`

e_1 und e_2 sind Ausdrücke. Die resultierende Eigenschaft ist erfüllt, wenn e_1 und e_2 den gleichen Wert liefern – der Vergleich wird dabei wie bei `check-expect` angestellt.

Abbildung 10.4 `expect`

Die Prozedur `number-list=?` funktioniert nur auf Listen von Zahlen. Es wäre möglich, `number-list=?` über der Vergleichsprozedur auf den Elementen zu abstrahieren, aber es wäre trotzdem umständliche Arbeit nur für den Zweck des Testens. Deshalb gibt es eine Vereinfachung analog zu `check-expect`. Die eingebaute Form `expect` akzeptiert zwei beliebige Werte und ist dann erfüllt, wenn diese Werte gleich sind. (Siehe Abbildung 10.4.) Die Eigenschaft von `invert` sieht damit so aus:

```
(check-property
 (for-all ((lis (list string)))
   (expect lis (invert (invert lis)))))
```

Viele Prozeduren auf Listen haben Eigenschaften, welche die Prozedur jeweils im Zusammenspiel mit einer oder mehreren anderen Prozeduren zeigen. Bei Prozeduren mit Listen ist es häufig interessant, das Zusammenspiel mit `concatenate` zu betrachten. Damit `concatenate` etwas sinnvolles tun kann, sind zwei Listen notwendig:

```
(check-property
  (for-all ((lis-1 (list number))
            (lis-2 (list number)))
    ...))
```

Auf diese zwei Listen kann `concatenate` aber auch jeweils `invert` angewendet werden:

```
(check-property
  (for-all ((lis-1 (list number))
            (lis-2 (list number)))
    ... (invert lis-1) ...
    ... (invert lis-2) ...
    ... (invert (concatenate lis-1 lis-2)) ...))
```

Wie läßt sich die Liste `(invert (concatenate lis-1 lis-2))` noch beschreiben? Angenommen, `lis-1` ist die Liste `#<list 1 2 3>` und `lis-2` die Liste `#<list 4 5 6>`. Dann gilt:

```
(invert (concatenate lis-1 lis-2))
=
(invert (concatenate #<list 1 2 3> #<list 4 5 6>))
```

$\Longrightarrow$ ... $\Longrightarrow$ (invert #<list $\underbrace{1\ 2\ 3}_{\text{lis-1}}$ $\underbrace{4\ 5\ 6}_{\text{lis-2}}$>))

$\Longrightarrow$ ... $\Longrightarrow$ #<list $\underbrace{6\ 5\ 4}_{\text{(invert lis-2)}}$ $\underbrace{3\ 2\ 1}_{\text{(invert lis-1)}}$ >

Dies läßt vermuten, daß die gesuchte Eigenschaft folgendermaßen aussieht:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number)))
   (expect (invert (concatenate lis-1 lis-2))
           (concatenate (invert lis-2) (invert lis-1)))))
```

Mantra 13 (Eigenschaften von Prozeduren auf Listen) Prozeduren, die Listen konsumieren, haben häufig interessante Eigenschaften im Zusammenspiel mit `concatenate`.

10.2.4 Eigenschaften von `list-sum`

`List-sum` aus Abschnitt 6.2 ist, wie `invert`, eine Prozedur, die eine Liste akzeptiert. Genau wie bei `invert` ist es eine gute Idee, die Interaktion zwischen `list-sum` und `concatenate` zu untersuchen. Es müssen also wieder zwei Listen her – die zu `invert` analoge Vorgehensweise liefert folgende Schablone:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number)))
    ... (list-sum lis-1) ...
    ... (list-sum lis-2) ...
    ... (list-sum (concatenate lis-1 lis-2)) ...
```

Da `list-sum` die Elemente der Liste addiert und die Addition assoziativ ist, müßte folgendes gelten:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number)))
    (expect (+ (list-sum lis-1) (list-sum lis-2))
            (list-sum (concatenate lis-1 lis-2)))))
```

Hier allerdings schlägt das Rundungsproblem aus Abschnitt 10.1.1 wieder zu: Die Addition auf `number` ist eben *nicht* assoziativ, aber immerhin auf `rational`. Der fertige Test muß also so aussehen:

```
(check-property
 (for-all ((lis-1 (list rational))
           (lis-2 (list rational)))
    (expect (+ (list-sum lis-1) (list-sum lis-2))
            (list-sum (concatenate lis-1 lis-2)))))
```

`Expect-within` liefert eine Eigenschaft analog zur Funktionsweise von `check-within`. Ein `expect-within`-Ausdruck hat folgende Form:

`(expect-within` e_1 e_2 e_δ`)`

e_1, e_2 und e_δ sind Ausdrücke, wobei e_δ eine reelle Zahl liefern muß. Die resultierende Eigenschaft ist erfüllt, wenn e_1 und e_2 den gleichen Wert liefern – der Vergleich wird dabei wie bei `check-within` angestellt, d.h. alle inexakten Zahlen in den Ergebnissen von e_1 und e_2 müssen nicht gleich sein, dürfen sich aber höchstens um e_δ voneinander unterscheiden.

Abbildung 10.5 `expect-within`

Eine Alternative ist die Verwendung der Form `expect-within`, die eine Eigenschaft analog zu `check-within` erzeugt. (Siehe Abbildung 10.5.)

Mit `expect-within` sieht der Testfall folgendermaßen aus:

```
(check-property
 (for-all ((lis-1 (list number))
           (lis-2 (list number)))
    (expect-within (+ (list-sum lis-1) (list-sum lis-2))
                   (list-sum (concatenate lis-1 lis-2))
                   0.1)))
```

Auch dieser Testfall läuft durch.

Auch der Test für die Assoziativität von + aus Abschnitt 10.1.1 kann mit `expect-within` formuliert werden:

```
(check-property
 (for-all ((a number)
           (b number)
           (c number))
    (expect-within (+ a (+ b c))
                   (+ (+ a b) c)
                   0.1)))
```

So wie sich die Assoziativität von + in einer Eigenschaft von `list-sum` niederschlägt, tut dies auch die Kommutativität: Sie besagt, daß die Reihenfolge der Elemente der Liste keine Rolle spielt. Eine einfache Möglichkeit, dies zu testen, ist wiederum mit zwei Listen zu arbeiten und diese einmal in der einen und dann in der anderen Richtung aneinanderzuhängen:

```
(check-property
 (for-all ((lis-1 (list rational))
           (lis-2 (list rational)))
   (expect (list-sum (concatenate lis-1 lis-2))
           (list-sum (concatenate lis-2 lis-1)))))
```

10.3 Eigenschaften von Prozeduren höherer Ordnung

In Abschnitt 9.5 wurde bereits eine Eigenschaft von `curry` und `uncurry` aufgeführt:

$$\texttt{(uncurry (curry } p\texttt{))} \equiv p$$

Mit anderen Worten: `curry` und `uncurry` sind jeweils Inverse voneinander. Auch diese Eigenschaft läßt sich direkt mit `check-property` und `for-all` formulieren. Zu beachten ist wieder, obwohl `curry` und `uncurry` polymorphe Verträge mit Vertragsvariablen haben, daß für den Test mit `check-property` ein „konkreter“ Vertrag ohne Vertragsvariablen für das Prozedur-Argument benutzt werden muß, also zum Beispiel `string`:

```
(check-property
 (for-all ((proc (string string -> string)))
   (expect (curry (uncurry proc))
           proc)))
```

Leider schlägt dieser Test fehl, und zwar mit einer mysteriösen Meldung:

```
Eigenschaft falsifizierbar mit #<procedure:?>
```

Offenbar ist DrScheme also der Ansicht, es hat eine Prozedur gefunden, welche die Eigenschaft nicht erfüllt, kann aber nicht genau sagen, welche Prozedur: Das liegt daran, daß es prinzipiell unmöglich ist, zwei Prozeduren auf Gleichheit zu überprüfen – Gleichheit zweier Prozeduren heißt ja, daß die eine Prozedur angewendet auf einen Satz Argumente *immer* das gleiche Ergebnis wie die andere Prozedur liefert. Im obigen Beispiel akzeptiert `proc` zwei Zeichenketten, von denen es unendlich viele gibt; die Gleichheit zu überprüfen, würde also unendlich lange dauern. `Expect` versucht es darum gar nicht erst, sondern sieht es als notwendige (nicht als hinreichende) Bedingung für die Gleichheit zweier Prozeduren an, daß sie Wert desselben `lambda`-Ausdrucks sind. `Expect` testet also bei Prozeduren auf sogenannte *intensionale Gleichheit*, was soviel heißt, daß `expect` vergleicht, *auf welche Weise* die beiden Prozeduren entstanden sind, nicht aber, ob sich die beiden Prozeduren *gleich verhalten*. Die letztere Eigenschaft heißt *extensionale Gleichheit* – und ist, wie gesagt, nicht effektiv testbar.

Der `lambda`-Ausdruck der Prozedur, die von `(curry (uncurry proc))` geliefert wird, ist aber der Rumpf von `curry`, während der `lambda`-Ausdruck von `proc` i.d.R. woanders steht; damit sind die beiden Prozeduren intensional ungleich, und der obige Test muß fehlschlagen, auch wenn die beiden Operanden von `expect` äquivalent sind.

Damit ein `check-property`-Test die Äquivalenz testen kann, muß er selbst `(curry (uncurry proc))` anwenden:

```
(check-property
 (for-all ((a string)
           (b string)
           (proc (string string -> string)))
   (expect ((uncurry (curry proc)) a b)
           (proc a b))))
```

Dieser Test funktioniert.

10.4 Programme beweisen

`Check-property` ist nützlich, um zu überprüfen, ob eine Eigenschaft gilt oder nicht. Da `check-property` allerdings nur eine endliche Menge zufälliger Tests durchführt, reicht es nicht aus, um sicherzugehen, daß eine bestimmte Eigenschaft für alle Werte der `for-all`-Variablen gilt: Dazu ist ein mathematischer Beweis notwendig.

An verschiedenen Stellen im Buch wurden Beweise für mathematische Funktionen durchgeführt – zuletzt in Abschnitt 6.5 für eine rekursive Funktion. Beweise über mathematische Funktionen erlauben, bei jedem Schrittan beliebigen Stellen Gleichungen einzusetzen. Beweise über Prozeduren in Programmen sind schwieriger, da sie das Substitutionsmodell berücksichtigen müssen: Bei jedem Reduktionsschritt kommt immer nur eine bestimmte Substitution in Frage.

10.4.1 Arithmetische Gesetze beweisen

Als erstes Beispiel für den Beweis an einem Programm dient der Beweis der Kommutativität von `+`. Der Beweis ist nicht besonders tiefgreifend, demonstriert aber die wichtigsten Techniken, die beim Beweisen von Programmen zum Einsatz kommen. Zu beweisen ist:

```
(= (+ a b) (+ b a))
```
$\Longrightarrow \ldots \Longrightarrow$ `#t`

$\ldots$ und zwar für beliebige Bindungen von `a` und `b` an Zahlen. Seien die Zahlen, die an `a` bzw. `b` gebunden sind, x und y mit $x,y \in \mathbb{C}$. (Die „mathematischen“ Namen könnten auch a und b sein, aber das birgt ein Verwirrungsrisiko mit `a` und `b`.) Wenn nun also der obige Term für bestimmte Werte von x und y im Substitutionsmodell reduziert wird, wird zuerst x für `a` und y für `b` eingesetzt. Für $x = 5$ und $y = 17$ also:

```
(= (+ 5 17) (+ 17 5))
```

Der Beweis soll aber für beliebige Werte für x und y funktionieren: x und y müssen also im Beweis auftauchen. Um den Unterschied zwischen Variablen des Programms `a` und `b` und den Zahlen zu machen, die für x und y eingesetzt werden, werden diese noch mit $\lceil\_\rceil$ umgeben: $\lceil x \rceil$ in einem Reduktionsschritt des Substitutionsmodell ist also ein Platzhalter für „die Zahl, für die x steht“ – entsprechend für y. Es gilt also:

```
(= (+ a b) (+ b a))
```
$=$ `(=` $\underline{\texttt{(+}\ \lceil x \rceil\ \lceil y \rceil\texttt{)}}$ `(+` $\lceil y \rceil\ \lceil x \rceil$`))`

Dort ist der erste Teilausdruck unterstrichen, der beim ersten Substitutionsschritt ersetzt wird. Wenn die Scheme-Prozedur `+` tatsächlich die mathematische Operation $+$ realisiert,[1] wird der Teilausdruck `(+` $\lceil x \rceil\ \lceil y \rceil$`)` durch $x+y$ ersetzt – beziehungsweise durch die Zahl, für die der mathematische Ausdruck $x+y$ steht. Es kommt also wieder $\lceil\_\rceil$ zum Einsatz:

[1] Die Komplikationen durch inexakte Zahlen und Rundungen bleiben hier unberücksichtigt.

(= $\underline{\text{(+ } \lceil x \rceil \ \lceil y \rceil \text{)}}$ (+ $\lceil y \rceil \ \lceil x \rceil$))
$\Longrightarrow$ (= $\lceil x+y \rceil$ $\underline{\text{(+ } \lceil y \rceil \ \lceil x \rceil \text{)}}$)

Entsprechend geht es weiter mit der zweiten Summe und schließlich der Vergleichsoperation =, die dem mathematischen $=$ entspricht:

$\Longrightarrow$ $\underline{\text{(= } \lceil x+y \rceil \ \lceil y+x \rceil \text{)}}$
$\Longrightarrow$ $\lceil x+y \ = \ y+x \rceil$
$=$ #t

Die Kommutativität der Scheme-Prozedur + folgt also aus der Kommutativität des mathematischen $+$, durch das sie definiert ist.

10.5 Rekursive Programme beweisen

Beweise über rekursive Programme sind anspruchsvoller als der Beweis der Kommutativität von +, benutzen aber die gleichen Techniken sowie – genau wie bei Beweisen mathematischer rekursiver Funktionen – Induktion als Beweisprinzip.

10.5.1 Rekursive Programme auf Listen

Als erstes Beispiel dient die Reflexivität. Es gilt für die Bindung von lis an eine beliebige Liste von Zahlen folgendes zu beweisen:

(number-list=? lis lis)
$\Longrightarrow \ldots \Longrightarrow$ #t

Wieder wird für den Wert der Bindung eine mathematische Variable eingeführt – l. Dann läuft der Beweis auf folgendes hinaus:

(number-list=? lis lis)
$=$ (number-list=? $\lceil l \rceil$ $\lceil l \rceil$)
$\Longrightarrow \ldots \Longrightarrow$ #t

Die ersten Reduktionsschritte sind:

(number-list=? $\lceil l \rceil$ $\lceil l \rceil$)
$\Longrightarrow$ ((lambda (lis-1 lis-2) ...) $\lceil l \rceil$ $\lceil l \rceil$)
$\Longrightarrow$ (cond ((empty? $\lceil l \rceil$) ...) ((pair? $\lceil l \rceil$) ...))

Damit es ab hier weitergeht, muß eine Fallunterscheidung für l gemäß der Datendefinition von Listen und analog zur Konstruktionsanleitung geben.

Angenommen, l ist die leere Liste. Dann geht es so weiter:

$\Longrightarrow$ (cond ((empty? $\lceil l \rceil$) ...) ((pair? $\lceil l \rceil$) ...))
$\Longrightarrow$ (cond (#t (cond ...)) ((pair? $\lceil l \rceil$) #f))

$\Longrightarrow$ `(cond ((empty?` $\lceil l \rceil$`) #t) ((pair?` $\lceil l \rceil$`) #f))`
$\Longrightarrow$ `(cond (#t #t) ((pair?` $\lceil l \rceil$`) #f))`
$\Longrightarrow$ `#t`

Für diesen Fall stimmt die Behauptung also. Angenommen, l ist *nicht* die leere Liste, hat also erstes Element f und Rest r. Dann geht die Reduktion folgendermaßen weiter:

$\Longrightarrow$ `(cond ((empty?` $\lceil l \rceil$`) ...) ((pair?` $\lceil l \rceil$`) ...))`
$\Longrightarrow$ `(cond (#f ...) ((pair?` $\lceil l \rceil$`) ...))`
$\Longrightarrow$ `(cond (#t (cond ...)))`
$\Longrightarrow$ `(cond ((empty?` $\lceil l \rceil$`) ...) ((pair?` $\lceil l \rceil$`) ...))`
$\Longrightarrow$ `(cond (#f ...) ((pair?` $\lceil l \rceil$`) ...))`
$\Longrightarrow$ `(cond (#t (and ...)))`
$\Longrightarrow$ `(and (=  (first` $\lceil l \rceil$`) (first` $\lceil l \rceil$`)) (number-list=? ...))`

Da `(first` $\lceil l \rceil$`)` das erste Element f liefert, geht es so weiter:

$\Longrightarrow$ `(and (=` $\lceil f \rceil$ `(first` $\lceil l \rceil$`)) (number-list=? ...))`
$\Longrightarrow$ `(and (=` $\lceil f \rceil$ $\lceil f \rceil$`) (number-list=? ...))`
$\Longrightarrow$ `(and #t (number-list=? ...))`
$\Longrightarrow$ `(number-list=? (rest` $\lceil l \rceil$`) (rest` $\lceil l \rceil$`))`

Da `(rest` $\lceil l \rceil$`)` den Rest r liefert, geht es dann so weiter:

$\Longrightarrow$ `(number-list=?` $\lceil r \rceil$ `(rest` $\lceil l \rceil$`))`
$\Longrightarrow$ `(number-list=?` $\lceil r \rceil$ $\lceil r \rceil$`)`

Weiter geht es unmittelbar nicht: Über r ist nichts bekannt. Natürlich wäre es möglich, wieder eine Fallunterscheidung wie bei l anzustellen, aber am Ende davon stünde wieder der gleiche Term, nur mit einem r' statt r, über das wiederum nichts bekannt ist.

Allerdings ist `number-list=?` eine *rekursive* Prozedur, und damit ist, wie auch bei rekursiven mathematischen Funktionen, Induktion anwendbar. Im Fall der leeren Liste ist die Behauptung bereits bewiesen. Im anderen Fall, also wenn l ein Paar aus f und r ist, ist die *Induktionsannahme* zulässig, also die, daß die Behauptung bereits auf dem Rest r gilt. Ausformuliert sieht die Induktionsannahme so aus:

`(number-list=?` $\lceil r \rceil$ $\lceil r \rceil$`)`
$\Longrightarrow \ldots \Longrightarrow$ `#t`

Das ist aber gerade der Schritt, der am Ende der obigen Reduktionskette fehlt: Damit ist der Beweis auch schon komplett. Das ist kein Zufall: Da die Prozedur der Konstruktionsanleitung folgt, folgt sie auch der induktiven Struktur von Listen, und der induktive Beweis ist leicht.

Für das nächste Mal sollte natürlich die Induktionsannahme bereits am Anfang des Beweises für den Fall „Paar“ formuliert werden, damit sofort klar ist, wenn die Annahme verwendet werden kann.

10.5.2 Rekursive Programme auf Zahlen

Die Definition von `factorial` am Anfang von Abschnitt 6.3 folgt der induktiven Definition der zugrundeliegenden Daten, der natürlichen Zahlen. Dementsprechend ist der Induktionsbeweis für dessen Korrektheit einfach. Es ist aber entscheidend, die zu beweisende Eigenschaft, welche die Korrektheit von `factorial` begründet, sorgfältig aufzuschreiben:

Die Prozedur `factorial` soll die Fakultät berechnen, es soll also für eine natürliche Zahl k gelten, falls die Variable `n` an k gebunden ist:

`(factorial n)`
$\Longrightarrow \ldots \Longrightarrow \lceil k! \rceil$

(Diese Eigenschaft läßt sich nicht sinnvoll mit `for-all` hinschreiben, da die mathematische Fakultät nicht fest eingebaut ist.)

Da es um natürliche Zahlen geht, gibt es zwei Fälle: $k = 0$ und $k > 0$. Zunächst $k = 0$:

`(factorial n)`
$=$ `(factorial` $\lceil k \rceil$`)`
$=$ `(factorial 0)`
$\Longrightarrow$ `((lambda (n) ...) 0)`
$\Longrightarrow$ `(if (= 0 0) ...)`
$\Longrightarrow$ `(if #t 1 ...)`
$\Longrightarrow$ `1`
$= \lceil 0! \rceil$

Im zweiten Fall ist $k > 0$, also gibt es eine natürliche Zahl l mit $k = l + 1$. Die Induktionsannahme ist, daß die Behauptung bereits für l bewiesen ist, also:

`(factorial` $\lceil l \rceil$`)`
$\Longrightarrow \ldots \Longrightarrow \lceil l! \rceil$

Der Beweis sieht so aus:

`(factorial n)`
$=$ `(factorial` $\lceil k \rceil$`)`
$\Longrightarrow$ `((lambda (n) ...)` $\lceil k \rceil$`)`
$\Longrightarrow$ `(if (=` $\lceil k \rceil$ `0) ...)`
$\Longrightarrow$ `(if #f 1 (* ...))`
$\Longrightarrow$ `(*` $\lceil k \rceil$ `(factorial (-` $\lceil k \rceil$ `1)))`
$\Longrightarrow$ `(*` $\lceil k \rceil$ `(factorial` $\lceil k-1 \rceil$`))`
$=$ `(*` $\lceil k \rceil$ `(factorial` $\lceil l \rceil$`))`

Mit der Induktionsannahme kann `(factorial` $\lceil l \rceil$`)` ersetzt werden:

`(*` $\lceil k \rceil$ `(factorial` $\lceil l \rceil$`))`
$\Longrightarrow \ldots \Longrightarrow$ `(*` $\lceil k \rceil$ $\lceil l! \rceil$`)`
$\Longrightarrow \lceil k \cdot l! \rceil$
$= \lceil k \cdot (k-1)! \rceil$
$= \lceil k! \rceil$

Damit ist der Beweis fertig.

Die Technik funktioniert auch mit Beispielen, bei denen die zu beweisende Eigenschaft nicht so einfach zu sehen ist wie bei `factorial`.

Die folgende Scheme-Prozedur verrät nicht auf den ersten Blick, was sie berechnet:

```
(: f (natural -> rational))

(define f
  (lambda (n)
    (if (= n 0)
        0
        (+ (f (- n 1))
           (/ 1 (* n (+ n 1)))))))
```

Tatsächlich berechnet der Prozeduraufruf `(f` $\lceil k \rceil$`)` für eine natürliche Zahl k die Zahl $\frac{k}{k+1}$.

Die Eigenschaft ist plausibel, wie sich mit `check-property` feststellen läßt:

```
(check-property
 (for-all ((k natural))
   (= (f k) (/ k (+ k 1)))))
```

Ein Beweis schafft Sicherheit. Wieder müssen die Fälle $k = 0$ und $k > 0$ betrachtet werden. Zunächst $k = 0$:

`(f` $\lceil k \rceil$`)`
$=$ `(f 0)`
$\Longrightarrow$ `((lambda (n) ...) 0)`
$\Longrightarrow$ `(if (= 0 0) 0 ...)`
$\Longrightarrow$ `(if #t 0 ...)`
$\Longrightarrow$ `0`
$= \lceil \frac{k}{k+1} \rceil$

Für den Fall $k > 0$ sei $l \in \mathbb{N}$ mit $k = l + 1$. Die Induktionsannahme ist die folgende:

`(f` $\lceil l \rceil$`)`
$\Longrightarrow \ldots \Longrightarrow \lceil \frac{l}{l+1} \rceil$

Dann gilt:

`(f` $\lceil k \rceil$`)`
$\Longrightarrow$ `((lambda (n) (if ...))` $\lceil k \rceil$`)`
$\Longrightarrow$ `(if (=` $\lceil k \rceil$ `0) ...)`
$\Longrightarrow$ `(if #f ... (+ ...))`
$\Longrightarrow$ `(+ (f (-` $\lceil k \rceil$ `1)) (/ 1 (*` $\lceil k \rceil$ `(+` $\lceil k \rceil$ `1))))`
$\Longrightarrow$ `(+ (f` $\lceil l \rceil$`) (/ 1 (*` $\lceil k \rceil$ `(+` $\lceil k \rceil$ `1))))`
$\Longrightarrow \ldots \Longrightarrow$ `(+` $\lceil \frac{l}{l+1} \rceil$ `(/ 1 (*` $\lceil k \rceil$ `(+` $\lceil k \rceil$ `1))))`
$\Longrightarrow$ `(+` $\lceil \frac{l}{l+1} \rceil$ `(/ 1 (*` $\lceil k \rceil$ $\lceil k+1 \rceil$`)))`

$\Longrightarrow$ (+ $\lceil \frac{l}{l+1} \rceil$ (/ 1 $\lceil k \cdot (k+1) \rceil$))
$\Longrightarrow$ (+ $\lceil \frac{l}{l+1} \rceil$ $\lceil \frac{1}{k \cdot (k+1)} \rceil$)
$\Longrightarrow$ $\lceil \frac{l}{l+1} + \frac{1}{k \cdot (k+1)} \rceil$
$= \lceil \frac{k-1}{k} + \frac{1}{k \cdot (k+1)} \rceil$
$= \lceil \frac{(k-1)(k+1)}{k \cdot (k+1)} + \frac{1}{k \cdot (k+1)} \rceil$
$= \lceil \frac{(k-1)(k+1) + 1}{k \cdot (k+1)} \rceil$
$= \lceil \frac{(k^2 - 1) + 1}{k \cdot (k+1)} \rceil$
$= \lceil \frac{k^2}{k \cdot (k+1)} \rceil$
$= \lceil \frac{k}{k+1} \rceil$

Damit die die Behauptung bewiesen.

10.6 Invarianten

Die bisher angewendete Technik für den Beweis rekursiver Prozeduren mit Induktion funktioniert bei Prozeduren mit Akkumulator nicht mehr direkt: Angenommen, die Korrektheit der endrekursiven Fakuktät `!` aus Abschnitt 8.1 soll ähnlich die die Korrektheit von `factorial` bewiesen werden. Wieder sei `n` an eine natürliche Zahl k gebunden:

(! n)
$=$ (! $\lceil k \rceil$)
$\Longrightarrow$ ((lambda (n) (!-helper n 1)) $\lceil k \rceil$)
$\Longrightarrow$ (!-helper $\lceil k \rceil$ 1)
$\Longrightarrow$ ((lambda (n acc) ...) $\lceil k \rceil$ 1)
$\Longrightarrow$ (if (= $\lceil k \rceil$ 0) 1 (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$)))

Wie bei `factorial` muß zwischen $k = 0$ und $k > 0$ unterschieden werden. Für $k = 0$ geht es folgendermaßen weiter:

$\Longrightarrow$ (if (= $\lceil k \rceil$ 0) 1 (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$)))
$\Longrightarrow$ (if #t 1 (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$)))
$\Longrightarrow$ 1

Für $k = 0$ funktioniert also der Beweis. Für $k > 0$ allerdings verläuft die Reduktion folgendermaßen:

$\Longrightarrow$ (if (= $\lceil k \rceil$ 0) 1 (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$)))
$\Longrightarrow$ (if #f 1 (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$)))
$\Longrightarrow$ (!-helper (- $\lceil k \rceil$ 1) (* 1 $\lceil k \rceil$))

Hier gibt es zwar einen rekursiven Aufruf mit Argument (- $\lceil k \rceil$ 1), aber *der Akkumulator hat sich auch verändert.* Damit ist die naheliegende Induktionsannahme für (!-helper (-

⌈*k*⌉ `1)` ⌈*a*⌉`)` (falls der Wert des Akkumulators `acc` *a* ist) wertlos. Prozeduren mit Akkumulator sind also nicht nur schwieriger zu schreiben als „regulär“ rekursive Prozeduren – sie sind auch schwerer zu beweisen.

Stattdessen ist es bei Prozeduren mit Akkumulator nützlich, eine *Invariante* aufzustellen, also eine Eigenschaft, welche Zwischenergebnis und noch zu leistende Arbeit in Beziehung setzt. Wie in Abschnitt 8.1 beschrieben, geht die Fakultätsprozedur mit Akkumulator folgendermaßen vor, um `(! 4)` auszurechnen:

$$(((1 \cdot 4) \cdot 3) \cdot 2) \cdot 1$$

Bei jedem rekursiven Aufruf läßt sich dieser Aufruf in „geleistete Arbeit“ (die durch den Akkumulator repräsentiert ist) und „noch zu leistende Arbeit“ unterteilen. Zum Beispiel entsteht ein rekursiver Aufruf `(!-helper 2 12)`, bei der Akkumulator der Wert des unterstrichenen Teilaufrufs ist:

$$(\underline{((1 \cdot 4) \cdot 3)} \cdot 2) \cdot 1$$

Es ist zu sehen, daß die noch zu leistende Arbeit gerade darin besteht, den Akkumulator noch mit der Fakultät von 2 zu multiplizieren. Wenn bei einem rekursiven Aufruf von `!-helper` der Wert von `n` *k* ist und der Wert des Akkumulators *a*, und am Ende die Fakultät von *N* berechnet werden soll, dann gilt bei jedem rekursiven Aufruf `(!-helper` ⌈*n*⌉ ⌈*a*⌉`)` immer:

$$a \cdot k! = N!$$

Dies ist die Invariante von `!-helper` und heißt so, weil sie beim rekursiven Aufruf von `!-helper` unverändert bleibt. Dies ist zunächst nur eine Behauptung, aber wenn sie gelten sollte, dann folgt daraus automatisch die Korrektheit der Prozedur, da bei $k = 0$ gilt:

$$a \cdot 0! = a \cdot 1 = a = N!$$

Daß $a \cdot k! = N!$ tatsächlich die Invariante ist, läßt sich folgendermaßen folgern:

- Sie gilt für den ersten Aufruf von `!-helper` von `!`, da dort $k = N$ und $a = 1$ gilt, also:

$$a \cdot k! = 1 \cdot N! = N!$$

- Jeder rekursive Aufruf erhält die Invariante. Angenommen, sie gilt für *k* und *a*, dann sind die neuen Werte für *k* und *a* im rekursiven Aufruf `(!-helper (-` ⌈*n*⌉ `1) (*` ⌈*a*⌉ ⌈*k*⌉`))` gerade $k \mapsto k-1$ und $a \mapsto a \cdot n$, die Invariante wäre also:

$$\begin{aligned}(a \cdot k) \cdot (k-1)! &= a \cdot (k \cdot (k-1)! \\ &= a \cdot k! \\ &= N!\end{aligned}$$

Diese Technik funktioniert auch bei weniger offensichtlichen Prozeduren. Hier eine Prozedur, die äquivalent ist zu der Prozedur `f` aus Abschnitt 10.5:

```
; n/(n+1) berechnen
(: f (natural -> natural))
```

```
(define f
  (lambda (n)
    (f-helper n 0)))

(define f-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (f-helper (- n 1)
                  (+ (/ 1 (* n (+ n 1)))
                     acc)))))
```

Die Prozedur geht folgendermaßen vor, um das Ergebnis für eine Eingabe n zu berechnen:

$$(\ldots((\frac{1}{n\cdot(n+1)}+\frac{1}{(n-1)\cdot((n-1)+1)})+\frac{1}{(n-2)\cdot((n-2)+1)})+\ldots+\frac{1}{1\cdot(1+1)})$$

Diese Summe ist bei jedem rekursiven Aufruf aufgeteilt als Summe von zwei Teilen, z.B. wie folgt:

$$(\ldots(\underline{(\frac{1}{n\cdot(n+1)}+\frac{1}{(n-1)\cdot((n-1)+1)})+\frac{1}{(n-2)\cdot((n-2)+1)})}+\ldots+\frac{1}{1\cdot(1+1)})$$

Der unterstrichene Teil ist gerade der Wert des Akkumulators, die Summe rechts davon der noch zu berechnende Summand. Wenn die Prozedur tatsächlich $n/(n+1)$ berechnen sollte, ist dieser rechte Teil im Beispiel $(n-3)/((n-3)+1)$. Damit ergibt sich die Invariante als $a+n/(n+1)$, wobei a der Wert von `acc` ist. Um die Annahme zu beweisen, daß dies die Invariante ist, muß im wesentlichen folgende Gleichung bewiesen werden:

$$a+\frac{n}{n+1}=a+\frac{n-1}{n}+\frac{1}{n\cdot(n+1)}$$

Dies ist eine lohnende Fingerübung.

Aufgaben

Aufgabe 10.1 Welche interessanten Eigenschaften hat die Division? Schreiben Sie diese als Eigenschaften von `/` in Scheme auf.

Aufgabe 10.2 Testen Sie den Isomorphismus von `curry` und `uncurry` aus Abschnitt 9.5 mit Hilfe von `check-property`.

Aufgabe 10.3 Schreiben Sie eine möglichst vollständige Liste interessanter Eigenschaften sowohl der Ihnen bekannten arithmetischen Operationen als auch der logischen Operationen auf. Finden Sie außerdem für jede Operation eine interessante Eigenschaft, die *nicht* gilt und überprüfen, ob DrScheme jeweils ein Gegenbeispiel findet.

Aufgabe 10.4 Versuchen Sie, die Eigenschafts-Tests für `number-list=?` auszutricksen, also eine fehlerhafte Version von `number-list=?` zu schreiben, die alle drei `check-property`-Tests besteht. Die `check-expect`-Tests sind für diese Aufgabe nicht relevant.

Aufgabe 10.5 Formulieren Sie Eigenschaften von `filter` und `map` im Zusammenhang mit `concatenate` und testen Sie diese.

Aufgabe 10.6 Finden Sie eine präzisere Formulierung der Kommutativität von `list-sum` als die in Abschnitt 10.2.4, also eine, an der sich die Eigenschaft, daß die „Reihenfolge der Elemente der Liste keine Rolle spielt“ klarer zu sehen ist.

Schreiben Sie dazu eine Prozedur, welche die Reihenfolge der Elemente einer Liste abhängig von einer natürlichen Zahl *n* verändert, z.B. indem die *n*te Permutation der Elemente ausgewählt wird.

Aufgabe 10.7 Schreiben Sie einen `check-property`-Test für folgende Eigenschaft:

$$\texttt{(uncurry (curry } p_2\texttt{))} \equiv p_2$$

Aufgabe 10.8 Formulieren Sie sinnvolle Eigenschaften von `compose` und `repeat` aus Abschnitt 9.4 und überprüfen Sie diese mit `check-property`!

Aufgabe 10.9 Beweise, daß für Prozeduren p_1 mit einem Parameter, die einparametrige Prozeduren zurückgeben, und Prozeduren p_2 mit zwei Parametern gilt:

$$\texttt{(curry (uncurry } p_1\texttt{))} \equiv p_1$$
$$\texttt{(uncurry (curry } p_2\texttt{))} \equiv p_2$$

Aufgabe 10.10 Beweisen Sie die entsprechend dem Beweis der Kommutativität von `+` in Abschnitt 10.4.1 die Assoziativität von `+` sowie die Distributivität von `+` und `*` aus Abschnitt 10.1.1.

Aufgabe 10.11 Beweisen Sie, daß die folgende Prozedur natürliche Zahlen quadriert:

```
; Quadrat einer Zahl berechnen
(: square (natural -> natural))
(define square
  (lambda (n)
    (if (= n 0)
        0
        (+ (square (- n 1))
           (- (+ n n) 1)))))
```

Formulieren Sie dazu auch eine Eigenschaft und überprüfen Sie diese mit `check-property`.

Aufgabe 10.12 Beweisen Sie, daß auch die folgende Prozedur `square` natürliche Zahlen quadriert. Geben Sie die Invariante von `square-helper` an!

```
; Quadrat einer Zahl berechnen
(: square (natural -> natural))
(define square
  (lambda (n)
    (square-helper n 0)))

(define square-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (square-helper (- n 1)
                       (+ acc
                          (- (+ n n) 1))))))
```

Formulieren Sie dazu auch eine Eigenschaft und überprüfen Sie diese mit `check-property`.