Christian Wagenknecht
Ronald Schaffhirt

D:\Texte\LinuxMagazin.doc (Rubens)

Stand: 18.03.01

Scheme als Mittel in der Lehre

Rechnen mit Unendlichem

Nicht nur dem am Streit über "die beste Programmiersprache" interessierten Leser wird aufgefallen sein, daß schon in den wenigen Linux-Magazin-Beiträgen zu Scheme sehr fortgeschrittene Themen, wie Prozeduren als Objekte erster Klasse, Makros und GUI-Programmierung, behandelt werden konnten. Dies ist bei anderen Programmiersprachen erst nach wesentlich mehr Vorbereitung möglich. Das für Scheme typische Abstraktionsniveau, s. z.B. continuations, bleibt im Allgemeinen sogar unerreicht.

Aufgrund der sehr einfachen Syntax einerseits und semantisch leistungsfähiger Sprachelemente andererseits eignet sich Scheme hervorragend als Formulierungsmittel abstrakter Konzepte. In der studentischen Ausbildung wird Scheme daher gerne als *didaktisches Vehikel* benutzt: "Repräsentiere es mit Scheme und spiele mit den definierten Funktionen herum, dann verstehst du besser, worum es geht."

In diesem Beitrag sollen zwei Beispiele für diesen didaktischen Ansatz vorgestellt werden. Der erste behandelt das *Rechnen mit unendlichen Objekten* und der zweite betrifft den Komplex *Web-Programmierung*. In beiden Fällen drücken wir unsere Ideen mit Scheme aus und benutzen das interaktive Programmieren mit Scheme als Mittel, nicht etwa als Gegenstand!

Darstellung unendlicher Objekte im endlichen Speicher

Als ziemlich selbstverständlich gilt die Feststellung, daß im Computer nur endliche Objekte bereitgehalten werden können. Beispielsweise werden rationale Zahlen im Allgemeinen als Festkommazahlen implementiert. Bei noch so großer Mantisse wird ein periodischer Dezimalbruch, wie $0.\bar{3}...$, ab einer bestimmten Nachkommastelle hemmungslos abgeschnitten. Die Pünktchen in der endlichen Schreibweise bedeuten ja, daß man nie aufhören darf, Dreien zu schreiben. Durch das Abschneiden sind nicht einmal die rationalen (geschweige denn die reellen) Zahlen computerrepräsentiert. Zwangsläufig wird anstelle mit den eigentlichen Größen nur mit entsprechenden Näherungswerten gearbeitet. Man beschränkt sich also auf Maschinenzahlen, für die einige der für rationale Zahlen geltenden Rechengesetze nicht mehr wirken. Klar, die Aufnahme unendlicher Objekt wird durch den stets endlichen Speicher begrenzt.

Nein, so klar ist das nicht! Wenn wir anstelle von $0.\bar{3}...$ den gleichwertigen Bruch $1/3$ betrachten, bietet sich eine *endliche* pünktchenfreie Darstellung für die gleiche Zahl. Diese Beobachtung führt zu der Idee, rationale und sogar reelle Zahlen durch ein *Verfahren zu deren Erzeugung* zu repräsentieren. Im Beispiel 1/3: "Dividiere 1 durch 3".

Achtung! *Es geht nicht um die Ausführung eines algorithmischen Verfahrens, sondern darum, eine Zahl durch ein zu ihrer Erzeugung anzuwendendes (ggf. nicht abbrechendes) Verfahren zu definieren*

Aber wie soll das Rechnen mit so repräsentierten Zahlen funktionieren? Wieviel ist beispielsweise $\sqrt{2}/\sqrt{3}$? Müssen hierzu die jeweiligen Verfahren addiert werden?

Datentyp "stream"

Im Folgenden führen wir einen abstrakten Datentyp "stream" (Strom oder Folge) ein, der wie folgt charakterisiert wird.

Ein *stream* ist ein Paar, dessen erstes Glied irgendein Scheme-Objekt (z.B. eine Ziffer) aber kein stream und dessen zweites Glied ein stream ist. Auf das erste Glied greifen wir mit `stream-car` und auf das zweite mit `stream-cdr` zu.

Dem Leser wird nachdrücklich empfohlen, an dieser Stelle von Fragen nach der Implementation von Sprachelementen für streams abzusehen. Wir nehmen an, alles Nötige sei vorhanden (oder eingebaut).

Zur Erzeugung eines konkreten streams verwenden wir einen Konstruktor `stream-cons`. Dieser erwartet zwei Argumente, nämlich genau die o.g. beiden Glieder des zu erzeugenden Paares.

Wir betrachten zwei Beispiel-streams:

```
(define integers
  (letrec
    ((integer-stream-maker
       (lambda (from)
         (stream-cons from (integer-stream-maker (+ from 1))))))
    (integer-stream-maker 0)))
```

Nun möchten wir uns 10 Elemente dieser Zahlenfolge ansehen.

```
> (stream-print integers 10)
```

schreibt

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

auf den Bildschirm.

Das zweite Beispiel betrifft die Folge der Fibonacci-Zahlen, nämlich

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Die `n`-te Fibonacci-Zahl ist durch folgende einfach Regel definiert: Die ersten beiden Fibonacci-Zahlen lauten `1`. Jede weitere ergibt sich aus der Summe der beiden vorhergehenden. Dies kann leicht als Scheme-Prozedur geschrieben werden.

```
(define fib
  (lambda (n)
    (if (< n 2)
        1
        (+ (fib (- n 1)) (fib (- n 2))))))
```

Berechnen Sie `(fib 29)` und beobachten Sie, wie lange Ihr Computer dafür benötigt.

Wir definieren die (unendliche) Folge der Fibonacci-Zahlen durch

```
(define fib-stream
  (letrec
    ((fib-stream-maker
       (lambda (from)
         (stream-cons (fib from) (fib-stream-maker (+ from 1))))))
    (fib-stream-maker 0)))
```

und lassen uns die ersten 30 Glieder dieser Zahlenfolge anzeigen:

```
> (stream-print fib-stream 30)
```

Dies nimmt erwartungsgemäß noch mehr Zeit in Anspruch als das weiter oben für `(fib 29)` der Fall war. Das Ergebnis ist

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
317811, 514229, 832040, ...
```

Evaluiert man nun den gleichen Ausdruck, also `(stream-print fib-stream 30)`, noch einmal, wird obiges Ergebnis ohne spürbaren Rechenzeitverbrauch ausgegeben. Dies ist ein willkommener Effizienzvorteil, der darauf zurückzuführen ist, daß berechnete Elemente eines streams nicht reevaluiert werden. *Scheme "merkt" sich bereits berechnete stream-Elemente.*

Evaluationskonzepte

Bevor wir nun mit streams weiterarbeiten, soll eine Begründung für dieses merkwürdige Evaluationsverhalten angegeben werden. Standardmäßig wird ein Scheme-Ausdruck der Form

```
(Operator Operand_1 Operand_2 ... Operand_n)
```

nach folgender Regel ausgewertet: Evaluiere alle Elemente der Liste und wende danach den Operator auf die Operanden an. Dabei ist die Reihenfolge der Evaluation der Teilausdrücke unbestimmt. Wichtig ist nur, daß die Operanden ausgewertet wurden, *bevor* der (evaluierte) Operator auf diese angewandt wird. Man spricht von *applicative-order* Evaluation, die gegenüber der *normal-order* Evaluation (von links nach rechts) Effizienzvorteile besitzt. Dies soll das folgende Beispiel zeigen.

```
  ((lambda (x) (x (x 5))) ((lambda (w) w) fib))
= ((lambda (x) (x (x 5))) fib)
= (fib (fib 5))
= (fib 8)
= 34
```

Normal-order Evaluation würde den Ausdruck `((lambda (w) w) fib)` zuerst auf der linken Seite (jeweils für `x`, also zweimal) einsetzen und den so entstandenen Ausdruck danach weiter reduzieren.

```
  ((lambda (x) (x (x 5))) ((lambda (w) w) fib))
= (((lambda (w) w) fib) (((lambda (w) w) fib) 5))
= (fib (fib 5))
= (fib 8)
= 34
```

Dies führt offensichtlich zu einem Effizienzverlust durch Mehrfachevaluation ein und desselben Teilausdrucks. In unserem Beispiel wird `((lambda (w) w) fib))` zweimal evaluiert.

Dieser Vorzug von applicative-order hat dazu geführt, daß diese Strategie in den gängigen Scheme-Systemen eingebaut ist. Die Effizienzvorteile wichtet man dabei höher als die Korrektheit der Reduktion besonders ausgefallener Scheme-Ausdrücke, wie

```
((lambda (x) 3)((lambda (x) (x x))(lambda (x) (x x)))),
```

die eher selten vorkommen. Die (applicative-order) Evaluation kommt nicht zum Ergebnis, obwohl doch die normal-order Evaluation terminiert:

```
  ((lambda (x) 3)((lambda (x) (x x))(lambda (x) (x x))))
= 3
```

Jedes `x` im Ausdruck `3` wird durch `((lambda (x) (x x))(lambda (x) (x x)))` ersetzt. Da der Ausdruck `3` kein einziges `x` enthält, ist nichts zu tun.

Die Standard-Scheme-Evaluation hat aber selbst ein Effizienzproblem, daß in anderen funktionsorientierten Sprachen, wie etwa Gofer, durch einen Strategiewechsel bekämpft wird. Der Wert des Ausdrucks

```
((lambda (w x y z) w) 1 (fib 29) (fib 29) (fib 29))
```

ist `1`. Die drei aufwendigen Berechnungen von `(fib 29)` sind völlig unnötig.

Dies führt zu der Idee, die Evaluation grundsätzlich erst dann durchzuführen, wenn der Wert des betrachteten Ausdruck auch wirklich gebraucht wird. Manchmal - so wie in obigem Beispiel - braucht man ihn nie. Diese Strategie nennt man *call-by-need*. Im Gegensatz zu der in Scheme standardmäßig implementierten *gierigen Evaluation* (eager evaluation) handelt es sich bei call-by-need um eine *verzögerte Evaluation* (*lazy evaluation*).

Zur Realisierung der verzögerten Evaluation in Scheme benötigt man lediglich zwei (eingebaute) Sprachelemente, nämlich

> `delay`, zur Erzeugung eines *verzögerten Ausdrucks* (promise genannt) und
> `force`, zur zwangsweisen Evaluation eines verzögerten Ausdrucks.

Vergleichen Sie die folgenden beiden Versionen für `crazy`

```
(define crazy
  (lambda (w x y z)
    x))

(define crazy-lazy
  (lambda (w x y z)
    (force x)))
```

```
> (crazy (fib 29)(fib 29)(fib 29)(fib 29))
832040

> (crazy-lazy
    (delay (fib 29)) (delay (fib 29))
    (delay (fib 29)) (delay (fib 29)))
832040
```

und interpretieren Sie die unterschiedlichen Rechenzeiten.

Verzögerte Evaluation hat noch einen weiteren Vorteil. Ein mit `force` zwangsevaluierter Ausdruck wird nicht reevaluiert, so wie wir das weiter oben bei der Arbeit mit streams bereits festgestellt hatten. Ein kleines Experiment unterstreicht diese Mitteilung.

```
> (define x (delay (fib 29)))
> (force x)
832040
> (force x)
832040
```

Die Berechnung von `(fib 29)` beim ersten `force` nimmt deutlich mehr Zeit in Anspruch als beim zweiten.

Implementation der Sprachelemente für streams

Nun ist alles vorbereitet, um die oben verwendeten Sprachelemente für die Arbeit mit streams zu implementieren. Da streams eine starke Verwandschaft zu (stets endlichen) Listen haben, verwenden wir eine Analogie.

	Liste	stream
Strukur	`(head . <liste>)`	`(head . <stream>)`
erstes Element	`(car <liste>)`	`(define stream-car car)`

Restliste	`(cdr <liste>)`	`(define stream-cdr`
		`  (lambda (stm)`
		`  (force (cdr stm))))`

Konstruktor	`(cons x <liste>)`	`stream-cons`
leeres Objekt	`'()`	`'()`
Prädikate	`list?` `null?`	`(define stream? pair?)` `(define stream-null? null?)`

Die Definition von `stream-cons` stellt uns vor ein Problem: Der Ansatz

```
(define stream-cons
  (lambda (head tail)
    ...))
```

ist nicht zielführend, denn beim Aufruf von `stream-cons` würde auch das zu verzögernde tail evaluiert werden. Wir beheben diese Schwierigkeit mit Hilfe einer Makrodefinition.

```
(define-syntax stream-cons
  (syntax-rules ()
    ((stream-cons head tail) (cons head (delay tail)))))
```

Ein weiteres nützliches Sprachelement für streams, das wir oben bereits verwendet haben, ist

```
(define stream-print
  (lambda (stm n)
    (cond
      ((= n 0) (printf "...~%"))
      (else
        (printf "~s, "(stream-car stm))
        (stream-print (stream-cdr stm) (- n 1))))))
```

`stream-print` dient zur Ausgabe der ersten `n` Elemente einer Folge. Interessiert man sich nur für das `n`-te Glied, leistet

```
(define stream-n-print
  (lambda (stm n)
    (if (= n 0)
        (printf "~s~%" (stream-car stm))
        (stream-n-print (stream-cdr stm) (- n 1)))))
```

gute Dienste.

Stream-Repräsentationen für reelle Zahlen

Für irrationale Zahlen $z = \sqrt{a}$, mit natürlichem $a \geq 2$, kann jeweils eine Intervallschachtelung (l_i, r_i) mit $l_i \leq z \leq r_i$ für $i \in N$ und rationalen Intervallgrenzen l_i, r_i angegeben werden.

(l_i, r_i) läßt sich beispielsweise mit der Halbierungsmethode konstruieren.

$(l_{j+1}, r_{j+1}) = \left(l_j, \frac{l_j + r_j}{2} \right)$, falls $a < \left(\frac{l_j + r_j}{2} \right)^2$, sonst $(l_{j+1}, r_{j+1}) = \left(\frac{l_j + r_j}{2}, r_j \right)$. Für die Anfangswerte gilt $l_0 = 1$ und $r_0 = a$.

Der Name der Prozedur `itvs` soll an Intervallschachtelung erinnern.

```
(define itvs
  (lambda (a)
    (letrec
      ((intervall-stream-maker
         (lambda (links rechts)
           (stream-cons
             (cons links rechts)
             (let ((mitte (/ (+ links rechts) 2)))
               (if (< a (* mitte mitte))
                   (intervall-stream-maker links mitte)
                   (intervall-stream-maker mitte rechts)))))))
      (intervall-stream-maker 1.0 a))))
```

Die (unendliche!) Intervallschachtelung `(itvs 2)` definiert also die reelle Zahl $\sqrt{2}$.

```
> (define qw2 (itvs 2))
> (stream-print qw2 10)
(1.0 . 2), (1.0 . 1.5), (1.25 . 1.5), (1.375 . 1.5), (1.375 . 1.4375),
(1.40625 . 1.4375), (1.40625 . 1.421875), (1.4140625 . 1.421875),
(1.4140625 . 1.41796875), (1.4140625 . 1.416015625), ...
```

Damit ist `qw2`$=\sqrt{2}$, obgleich wir beim Betrachten (mit `stream-print` oder `stream-n-print`) der rationalen Intervalle für hinreichend großes `i` nur Näherungswerte erhalten.

Zur Definition rationaler/irrationaler Zahlen werden auch abbrechende/nichtabbrechende Kettenbruchentwicklungen verwendet. In der Theorie gibt es weitere konstruktive Verfahren, mit denen reelle Zahlen definiert werden können.

Wichtig ist, daß wir mit der reellen Zahl `qw2`$=\sqrt{2}$ mit Scheme (exakt!) rechnen können.

Rechnen mit stream-repräsentierten Zahlen

Wir führen das für die Division der irrationalen Zahlen $x = \sqrt{k_1}$ und $y = \sqrt{k_2}$, mit $a_i^l \le x \le a_i^r$ bzw. $b_i^l \le x \le b_i^r$, vor. Wegen $a_i^l \le x \le a_i^r$ und $\frac{1}{b_i^r} \le \frac{1}{y} \le \frac{1}{b_i^l}$ gilt $\frac{a_i^l}{b_i^r} \le \frac{x}{y} \le \frac{a_i^r}{b_i^l}$.

Außerdem kann man zeigen, daß die Intervalllängen beliebig klein werden.

Aus diesen theoretischen Überlegungen ergibt sich die folgende Scheme-Prozedur unmittelbar. Das Ergebnis ist ein stream!

```
(define itv/
  (lambda (stm1 stm2)
    (stream-cons
      (let ((head1 (stream-car stm1))(head2 (stream-car stm2)))
        (cons (/ (car head1)(cdr head2))(/ (cdr head1)(car head2))))
      (itv/ (stream-cdr stm1)(stream-cdr stm2)))))
```

Wir berechnen exemplarisch $\sqrt{2}/\sqrt{3}$, d.h.

```
> (define qw2/qw3 (itv/ qw2 (itvs 3)))
```

und schauen uns das zwanzigste Intervall an.

```
> (stream-n-print qw2/qw3 20)
(0.8164958693703516 . 0.8164973191071839)
```

Man kann also mit den definierten unendlichen Objekten rechnen.

Die Behandlung der Summe $\sqrt{2}+\sqrt{3}$ überlassen wir dem Leser als Übungsaufgabe.

Sollten Sie sich am Potenzieren der hier betrachteten Zahlen versuchen wollen, ist zu beachten, daß die Intervallgrenzen nicht rational bleiben.

Sieb des Eratosthenes

Zwei abschließende Beispiele aus dem Bereich der Zahlenfolgen veranschaulichen die Leistungsfähigkeit des beschriebenen Konzepts. Zunächst geht es um die Menge der Primzahlen, die als (unendliche) Zahlenfolge definiert werden kann:

Nimm die Folge der natürlichen Zahlen ab 2:

```
2, 3, 4, 5, 6, 7, ... = (stream-cdr (stream-cdr integers))
```

Filtere alle Vielfachen des jeweils ersten Gliedes der Zahlenfolge aus:

```
    2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ...
= (filtere-aus (lambda (x)(teilt? 2 x) (stream-cdr (stream-cdr integers)))
    2, 3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, ...
= (filtere-aus (lambda (x)(teilt? 3 x) (stream-cdr (stream-cdr integers)))
```

usw. Die übrig bleibenden Zahlen sind genau die Primzahlen.

```
(define teilt?
  (lambda (t n)
    (zero? (remainder n t))))

(define filtere-aus
  (lambda (praed stm)
    (if (praed (stream-car stm))
        (filtere-aus praed (stream-cdr stm))
        (stream-cons
          (stream-car stm)
          (filtere-aus praed (stream-cdr stm))))))

(define sieb
  (lambda (stm)
    (stream-cons
      (stream-car stm)
      (sieb
        (filtere-aus
          (lambda (x) (teilt? (stream-car stm) x))
          (stream-cdr stm))))))
```

Die Folge der Primzahlen ergibt sich aus

```
> (define primzahlen (sieb (stream-cdr (stream-cdr integers))))
```

Wir lassen uns die ersten 100 Primzahlen auf den Bildschirm ausgeben:

```
> (stream-print primzahlen 100)
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, ...
```

Folge der Quotienten benachbarter Fibonacci-Zahlen

Berechnet man die Quotienten jeweils benachbarter Fibonacci-Zahlen stellt man fest, daß sich ein bestimmter Wert, nämlich 1.61803... einzustellen scheint. Bevor man den entsprechenden Grenzwert berechnet, hilft die Quotientenfolge bei der Gewinnung der angegebenen Hypothese.

```
(define fibquot
  (lambda (stm)
    (stream-cons
      (/ (stream-car (stream-cdr stm))
       (fixnum->flonum (stream-car stm)))
      (fibquot (stream-cdr stm)))))

> (stream-print (fibquot fib-stream) 18)
1.0, 2.0, 1.5, 1.6666666666666667, 1.6, 1.625, 1.6153846153846154,
1.619047619047619, 1.6176470588235294, 1.6181818181818182,
1.6179775280898876, 1.6180555555555556, 1.6180257510729614,
1.6180371352785146, 1.618032786885246, 1.618034447821682,
1.6180338134001253, 1.618034055727554, ...
```

Zusammenfassung

Mit Hilfe von streams können unendliche Objekte definiert, gespeichert und verarbeitet werden. Die Komponenten dieser Objekte (Intervalle, Folgenglieder, ...) bilden stets eine *potentiell unendliche* oder *abzählbare Menge*. Im Gegensatz zu überabzählbaren Mengen haben abzählbare genau so viele Elemente wie die Menge der natürlichen Zahlen.

Die gesamte Menge der reellen Zahlen kann im Computer nicht gespeichert werden.

Damit haben wir doch recht abstrakte Einsichten gewonnen. Die entwickelten und eingesetzten Scheme-Prozeduren haben den Sachverhalt konkretisiert und somit zum Verständnis beigetragen.