

Kapitel 11

Definitionale Interpretation und Continuations

Der Interpreter des letzten Kapitels ist eine präzise Beschreibung des Umgebungsmodells: Alles, was in Kapitel 8 über das Umgebungsmodell gesagt wurde, steht in diesem Interpreter codiert. Damit stellt er eine Art *Semantik* dar — er weist einem Mini-Scheme-Programm eine Bedeutung zu, indem er beschreibt, wie es ausgewertet wird. Es wäre nun praktisch, wenn solch ein Interpreter als Sprach-*Definition* verwendbar wäre: Alle Auslegungs- und Zweifelsfragen ließen sich anhand der Interpreters klären. Wer den Interpreter versteht, versteht auch die Sprache.

Leider läßt der Interpreter einige Fragen über die Auswertung von Mini-Scheme-Programmen unbeantwortet: Das liegt daran, daß er in Teilen „schummelt“ und ein Objektkonstrukt einfach durch das entsprechende Metakonstrukt realisiert: Wer das `if` in Scheme schon nicht versteht, wird es in Mini-Scheme auch nicht verstehen. Es scheint also, als ob für die Definition einer Sprache metazirkuläre Interpretation nicht der richtige Mechanismus ist: eine andere, einfachere, genau verstandene Sprache scheint notwendig. Zum Glück läßt sich als diese einfachere Sprache auch wieder Scheme verwenden, aber in einer „abgespeckten Version“. Wenn dem Interpreter unterschiedliche Grade der Askese auferlegt werden, bringt er mehr oder weniger Bedeutung von Mini-Scheme-Programmen zum Vorschein.

11.1 Weniger Bedeutung: Prozeduren mit Metaprozeduren

Vielleicht ist es am einfachsten, zuerst eine Variante des Interpreters zu betrachten, der *weniger* über das Umgebungsmodell aussagt als der Code aus dem letzten Kapitel. Eine Möglichkeit, dies zu tun, liegt in der Repräsentation von Prozeduren. Das Umgebungsmodell schreibt die Verwendung von Closures vor. Es ist jedoch möglich, Mini-Scheme-Prozeduren im Interpreter durch Scheme-Prozeduren zu repräsentieren. Zum Vergleich hier noch einmal der alte Code mit Closure-Erzeugung:

```
((lambda? expression)
 (make-ordinary-value
  (make-closure (lambda-parameters expression)
                (lambda-body expression)
                environment)))
```

Die Closure wird später ausgepackt, damit die Prozedur angewendet werden und die Auswertung im Rumpf fortfahren kann. Der entsprechende Code in `apply-procedure` im alten Interpreter sieht folgendermaßen aus:

```

(let* ((closure (ordinary-value procedure))
      (parameters (closure-parameters closure))
      (new-frame
       (make-frame (zip parameters parameter-values)))
      (environment
       (make-environment new-frame
                        (closure-environment closure))))
      (evaluate (closure-body closure) environment))

```

Genau dieser Code läßt sich aber in den Rumpf einer Abstraktion stecken:

```

((lambda? expression)
 (let ((parameters (lambda-parameters expression))
      (body (lambda-body expression)))
  (make-ordinary-value
   (lambda (parameter-values)
     (let* ((new-frame
            (make-frame (zip parameters parameter-values)))
           (environment
            (make-environment new-frame environment)))
       (evaluate body environment))))))

```

Der entsprechende Code in `apply-procedure` wird dramatisch vereinfacht:

```

((ordinary-value procedure) parameter-values)

```

Was ist gewonnen? Der Code ist insgesamt kürzer geworden. (Die ganzen Definitionen, die mit Closures zu tun haben, können ersatzlos gestrichen werden.) Leider verrät der entstandene Interpreter nichts mehr über die Repräsentation von Prozeduren, und damit fast überhaupt nichts mehr darüber, wie Prozeduren und Prozeduranwendung allgemein funktionieren. Auch die Funktionsweise der lexikalischen Bindung bleibt ein Mysterium. (Das Spiel mit der metazirkulären Interpretation läßt sich noch weiter treiben. Nicht hier.)

11.2 Wie funktioniert Prozedurrückkehr?

Zurück zum ursprünglichen Interpreter des letzten Kapitels: Dieser erklärt die Repräsentation von Prozeduren. Doch erklärt er damit alles, was es über Prozedurauf-rufe zu wissen gibt? Das Problem läßt sich beim Laufenlassen des Interpreters auf dem Fakultätsbeispiel sehen. Hier noch einmal der Code:

```

(define fac
  (lambda (n)
    (if (= n 1)
        1
        (* n (fac (- n 1))))))

```

Hier der Ablauf:

```

(apply-procedure ⟨n, (if (= n 1) 1 (* n (fac (- n 1))))⟩, []) (4)
⇒ (evaluate (if (= n 1) 1 (* n (fac (- n 1)))) [n ↦ 4])
⇒ ...
⇒ (* 4 (apply-procedure ⟨n, (if (= n 1) 1 (* n (fac (- n 1))))⟩, []) (3))
⇒ ...
⇒ (* 4 (* 3 (apply-procedure ⟨n, (if (= n 1) 1 (* n (fac (- n 1))))⟩, []) (2))
...

```

Diese Auswertung bezieht sich auf den Interpreter, findet also auf der Metaebene statt: Es ist deutlich sichtbar, daß sich auch hier Kontext anstaut. Dieser Kontext entspricht aber keinem Datenobjekt, das durch den Interpreter propagiert wird: Vielmehr wird der vermutete „Kontextspeicher“ des Scheme-Systems verwendet, über den nichts bekannt ist. Damit läßt auch der Interpreter mit Closures die Frage unbeantwortet, was eigentlich *wirklich* bei der Prozedurrückkehr passiert. (Bei näherer Betrachtung wird sogar deutlich, daß der Interpreter einige Fragen über die Auswertungsreihenfolge unbeantwortet läßt.) Es ist also im Interesse der Klarheit wünschenswert, einen Interpreter zu konstruieren, der diesen Kontext sichtbar macht.

11.3 Continuation-Passing Style

Das Problem mit dem Kontext läßt sich auch ohne Interpreter betrachten. Als Beispiel dient — wieder einmal — die Fakultätsfunktion:

```
(define fac
  (lambda (n)
    (if (= n 1)
        1
        (* n (fac (- n 1))))))
```

Bereits das Substitutionsmodell zeigt, daß `fac` um die rekursiven Aufrufe von sich selbst jedesmal eine Multiplikation wickelt und damit Kontext generiert:

```
(fac 4)
=> (* 4 (fac 3))
=> (* 4 (* 3 (fac 2)))
=> (* 4 (* 3 (* 2 (fac 1))))
```

Diese Akkumulation von Kontext geschieht im Umgebungsmodell auf die gleiche Art und Weise. Das Erklärungsproblem hat zwei Teile: Welcher Teilausdruck von

```
(* 4 (* 3 (* 2 (fac 1))))
```

soll zuerst ausgewertet werden, und was passiert danach? Das erste Problem ist durch die Auswertungsregel für Prozeduraufrufe erklärt: Die Operanden eines Prozeduraufrufs werden vor dem Prozeduraufruf selbst ausgewertet, was zu einer Auswertung „von innen nach außen“ führt. Was aber passiert *danach*, mit dem Wert von `(fac 1)`? Diese Information steckt im Kontext von `(fac 1)`, der sich so aufschreiben läßt:

```
(* 4 (* 3 (* 2 [])))
```

Dort also, wo das `(fac 1)` stand, wird ein „Loch“ in den Ausdruck geschnitten. Das aber ist intuitiv genau das, was ein Kontext ist: ein „Ausdruck mit Loch“, der darauf wartet, daß das Loch mit einem Wert gefüllt wird. Die leeren eckigen Klammern `[]` sind die übliche Notation für das Loch, aber es ginge auch mit einem Namen:

```
(* 4 (* 3 (* 2 result)))
```

Tatsächlich läßt sich dieser Kontext als Prozedur repräsentieren:

```
(lambda (result) (* 4 (* 3 (* 2 result))))
```

Hieße diese Prozedur `k`, könnte die Berechnung der Fakultät von 4 folgendermaßen beschrieben werden:

```
(k (fac 1))
⇒
(* 4 (* 3 (* 2 (fac 1))))
```

Mit diesem Wissen gewappnet läßt sich eine neue Version von `fac` schreiben, die den Kontext immer in Form einer Prozedur mit sich trägt:

```
(define fac-cps
  (lambda (n k)
    (if (= n 1)
        (k 1)
        (fac-cps (- n 1)
                  (lambda (result)
                    (k (* n result)))))))
```

`Fac-cps` hat gegenüber `fac` einen zusätzlichen Parameter `k`, der den Kontext re-präsentiert. Es läßt sich leicht beweisen, daß

$$(k (fac n)) \equiv (fac-cps n k)$$

insbesondere also:

$$(fac n) \equiv (fac-cps n (lambda (result) result))$$

`Fac-cps` jedoch verhält sich bei der Auswertung anders als `fac`:

```
(fac-cps 4 k)
⇒
(fac-cps 3 (lambda (result) (k (* 4 result))))
⇒
(fac-cps 2 (lambda (result)
              ((lambda (result)
                 (k (* 4 result)))
               (* 3 result))))
⇒
(fac-cps 1 (lambda (result)
              ((lambda (result)
                 ((lambda (result)
                    (k (* 4 result)))
                  (* 3 result)))
               (* 2 result))))
⇒
((lambda (result)
   ((lambda (result)
      (k (* 4 result)))
    (* 3 result)))
 (* 2 1))
⇒
((lambda (result)
   (k (* 4 result)))
 (* 3 2))
⇒
(k (* 4 6))
```

Die neue Fassung der Fakultät ist endrekursiv und akkumuliert damit *keinen* Kontext! Endrekursion ist nun aber ein primitiverer Mechanismus als normale Rekursion. (Auf Maschinenebene läßt sich Endrekursion durch einfache Sprünge abbilden.)

Fac-cps ließe sich auch in einem hypothetischen Scheme-System auswerten, daß keine Maschinerie für das Abspeichern von Kontexten besitzt.

Der Kontext ist natürlich nicht ersatzlos verschwunden: Der Wert des Parameters *k* — die Repräsentation des Kontexts — wird bei jedem rekursiven Aufruf „größer“ und verschlingt damit Speicherplatz. In dieser Beziehung ist durch den Schritt nichts gewonnen, dafür aber ist der Kontext von der Metaebene auf die Objektebene transferiert worden und läßt sich dort studieren und manipulieren.

Der Programmierstil von fac-cps — mit expliziter Repräsentation des Kontextes als Prozedur — heißt *Continuation-Passing Style*, die Repräsentation des Kontextes als Prozedur entsprechend *Continuation*. Die Continuation ist die *Zukunft* des Wertes eines Ausdrucks. Continuations sind der Schlüssel zum Verständnis von Kontrollstrukturen. Die Umwandlung einer „normalen“ Prozedur in CPS läßt sich automatisieren mit Hilfe der sogenannten *CPS-Transformation*, eines der zentralen Erkenntnisse des Compilerbaus. In diesem Kapitel wird allerdings auch weiterhin Handarbeit angewendet.

11.4 Mehr Bedeutung: Interpreter in CPS

Ebenso wie fac lassen sich auch evaluate und apply-procedure in CPS übertragen. (Es ginge auch mit dem Rest des Interpreters, aber der geistige Nährwert steckt in diesen beiden Prozeduren.) Zunächst einmal bekommt evaluate einen zusätzlichen Parameter:

```
(define evaluate
  (lambda (expression environment k)
    (cond
```

Bei Literalen und Variablen muß darauf geachtet werden, daß jeweils die Continuation mit dem Ergebnis aufgerufen wird:

```
((literal? expression)
 (k
  (make-ordinary-value (literal-constant expression))))
((variable? expression)
 (k
  (environment-lookup environment
    (variable-name expression))))
```

Bei Conditionals wird der Test ausgewertet mit einer Continuation, in der zwischen Konsequente und Alternative entschieden wird:

```
((if? expression)
 (evaluate (if-test expression) environment
  (lambda (test-value)
    (if (ordinary-value test-value)
        (evaluate (if-consequent expression) environment k)
        (evaluate (if-alternative expression) environment k)))))
```

Bei Zuweisungen wird zunächst die rechte Seite ausgewertet. In der Continuation wird dann die Umgebung entsprechend verändert:

```
((set!? expression)
 (evaluate (set!-expression expression)
  environment
  (lambda (rhs-value)
```

```

(environment-mutate-binding! environment
  (set!-variable-name expression)
  rhs-value)
(k unspecific-value))))

```

Bei `begin` wird es etwas technisch: Der rekursive Aufruf steckt in der Continuation, `k` wird ganz am Schluß aufgerufen:

```

((begin? expression)
 (letrec ((loop (lambda (expressions last-value)
  (if (null? expressions)
    (k last-value)
    (evaluate (car expressions) environment
      (lambda (value)
        (loop (cdr expressions) value)))))))
 (loop (begin-expressions expression) #f)))

```

Abstraktionen sind wieder einfacher: Es wird eine Closure erzeugt wie gehabt, diese wird an `k` übergeben:

```

((lambda? expression)
 (k
  (make-ordinary-value
   (make-closure (lambda-parameters expression)
    (lambda-body expression)
    environment))))

```

Der Code für Prozeduranwendungen wird dadurch kompliziert, daß mehrere Teilausdrücke — Operator und Operanden — ausgewertet werden müssen und dementsprechend mit Continuations hantiert wird:

```

(evaluate
 (application-operator expression)
 environment
 (lambda (procedure)
  (letrec ((loop
    (lambda (operands reverse-values)
     (if (null? operands)
      (apply-procedure procedure (reverse reverse-values)
        k)
      (evaluate (car operands) environment
        (lambda (operand-value)
          (loop (cdr operands)
            (cons operand-value
              reverse-values))))))))
 (loop (application-operands expression) '()))))

```

`Apply-procedure` fängt bei den Primitiva ganz harmlos an:

```

(define apply-procedure
 (lambda (procedure parameter-values k)
  (cond
 ((value-primitive? procedure)
  (k
   (make-ordinary-value
    (apply (primitive-value procedure)
     (map ordinary-value parameter-values))))))

```

Bei „echten“ Prozeduranwendungen ist der Code ebenfalls weitgehend unverändert, mit dem einzigen Unterschied, daß `k` wieder an den rekursiven Aufruf von `evaluate` weitergereicht wird:

```
((value-ordinary? procedure)
  (let* ((closure (ordinary-value procedure))
        (parameters (closure-parameters closure))
        (new-frame
         (make-frame (zip parameters parameter-values)))
        (environment
         (make-environment new-frame
                           (closure-environment closure))))
    (evaluate (closure-body closure) environment k)))
```

Damit ist der Kern des Interpreters fertig: Der Kontext wird explizit weitergereicht, alle rekursiven Aufrufe sind endrekursiv.

Noch müssen die beiden Prozeduren `evaluate-program` und `process-definition` geändert werden, die beide `evaluate` aufrufen. Dabei wird als Continuation jeweils der leere Kontext

```
(lambda (x) x)
```

benutzt, sonst ändert sich nichts.

```
(define evaluate-program
  (lambda (forms)
    (let ((global-environment (make-global-environment)))
      (letrec
        ((loop
         (lambda (forms reverse-values)
           (cond
            ((null? forms) (reverse reverse-values))
            ((define? (car forms))
             (process-definition (define-variable-name (car forms))
                                (define-expression (car forms))
                                global-environment)
             (loop (cdr forms) reverse-values))
            (else
             (let ((value (evaluate (car forms) global-environment
                                   (lambda (x) x))))
               (loop (cdr forms) (cons value reverse-values)))))))
         (map ordinary-value
              (loop forms '()))))))))
```

```
(define process-definition
  (lambda (name expression environment)
    (let ((value (evaluate expression environment (lambda (x) x))))
      (extend-frame!
       (environment-frame environment)
       name
       value))))
```

11.5 Continuations an der Oberfläche

Durch die Umwandlung ins CPS sind Continuations in der Hierarchie aus Scheme-System, Interpreter und Mini-Scheme-Programm eine Stufe höher gewandert. In

der Tat lassen sich Continuations noch eine weitere Stufe nach oben treten, so daß Mini-Scheme-Programme auf sie zugreifen können. Dazu wird allerdings ein weiteres Sprach-Element notwendig, daß die Continuation als einen Wert des Mini-Scheme-Programms repräsentiert. Es heißt `let/cc` für „let-current-continuation“:

```
⟨expression⟩ ::= ⟨let/cc⟩
⟨let/cc⟩ ::= (let/cc ⟨variable⟩ ⟨expression⟩)
```

Die Idee ist, daß bei der Auswertung eines Ausdrucks

```
(let/cc v e)
```

die momentane Continuation als eine *Escape-Prozedur* repräsentiert wird, die an *v* gebunden wird. Diese Escape-Prozedur hat einen Parameter. Wenn sie aufgerufen wird, wird die ursprüngliche Continuation des `let/cc`-Ausdrucks zur neuen momentanen Continuation. Folgendes Programm demonstriert rudimentär die Arbeitsweise von `let/cc`:

```
(define demo-context #f)
(+ (let/cc k
    (begin
      (set! demo-context k)
      (+ 23 42)))
  4)
(demo-context 27)
```

Der letzte Ausdruck, `(demo-context 27)` produziert den Wert 31.

`Let/cc` ist im CPS-Interpreter erstaunlich einfach zu realisieren. Zunächst die Syntax:

```
(define let/cc? (make-combination-predicate 'let/cc))

(define let/cc-name
  (lambda (form)
    (car (cdr form))))

(define let/cc-expression
  (lambda (form)
    (car (cdr (cdr form)))))
```

Hauptzutat ist eine neue Sorte Wert für die Escape-Prozeduren, zusätzlich zu primitiven und gewöhnlichen Werten:

```
(define continuation-type (make-type 'continuation-value))

(define make-continuation
  (lambda (value)
    (make-typed-object continuation-type value)))

(define value-continuation?
  (typed-object-predicate continuation-type))

(define continuation-value
  (lambda (value)
    (typed-object-value continuation-type value)))
```

Let/cc legt k in einen solchen continuation-Wert und plaziert die entsprechende Bindung in ein neues Frame. Der neue cond-Fall in evaluate sieht folgendermaßen aus:

```
((let/cc? expression)
  (let* ((continuation (make-continuation k))
        (frame (make-frame (list (cons (let/cc-name expression)
                                       continuation))))
        (environment
         (make-environment frame environment)))
    (evaluate (let/cc-expression expression) environment
              k)))
```

Apply-procedure muß erweitert werden um einen Fall für continuation-Werte:

```
((value-continuation? procedure)
  ((continuation-value procedure) (car parameter-values)))
```

Fertig ist let/cc! Let/cc ist in der Lage, *alle* Kontrollstrukturen (in einem gewissen theoretischen Sinne) nachzubilden — etwa Exceptions, Nebenläufigkeit, Nichtdeterminismus, Resolution etc. Tatsächlich enthält auch normales Scheme eine Variante von let/cc in Form einer Prozedur namens `call-with-current-continuation`, die in Mini-Scheme folgendermaßen definiert wäre:

```
(define call-with-current-continuation
  (lambda (p)
    (let/cc k
      (p k))))
```

Das obige Beispiel würde in Scheme mit `call-with-current-continuation` zu:

```
(define demo-context #f)
(+ (call-with-current-continuation
   (lambda (k)
     (begin
       (set! demo-context k)
       (+ 23 42))))
  4)
(demo-context 27))
```