Programmierung und Modellierung

Termauswertung

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

Inhalt

- Wiederholung: Programmiermethodik für rekursive Funktionen
  - Wohlfundierte Relation
- Kap. 3 Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)
  1. Auswertung von Ausdrücken
  2. Auswertung in applikativer und in normaler Reihenfolge
  3. Verzögerte Auswertung
  4. Auswertung der Sonderausdrücke

Wiederholung: (Rekursive) Funktionen

Beispiel: Konstante, Quadrierung, schnelle Potenzbildung

```
- val zwei = 2;
val zwei = 2 : int

- fun quadrat(x:int) = x*x;
val quadrat = fn : int -> int

- fun exp(x: int, n) =
    if n=0 then 1
    else if  n mod 2 = 0 then quadrat(exp(x, n div 2))
         else x * quadrat(exp(x, n div 2));
val exp = fn : int * int -> int
```

Wohlfundierte Relationen

- Sei M eine Menge.

  Eine Relation $R \subseteq M \times M$ heißt **wohlfundiert**, wenn jede absteigende Kette endlich ist; d.h. wenn es

  keine unendliche Folge $a_1, a_2, a_3, \ldots$ von Elementen in M gibt,

  so dass $a_{i+1} \, R \, a_i$ für alle $i \in N$.
- Beispiele
  - Kleiner-Relation auf natürlichen Zahlen
    - M = N und xRy gdw. $x < y$
  - Lexikographische Ordnung
- Ist R eine wohlfundierte Relation auf einer Menge M, so kann man anstelle einer Abstiegsfunktion m : A –> N auch eine

  Abstiegsfunktion m : A –> M

  wählen, derart dass immer

  m(y) R m(x) gilt, wenn `f(y)` im Rumpf `E(f, x)` aufgerufen wird.
  - Beispiel: Abstiegsfkt. von exp(x,n): m(x,n) = n; R Kleiner-Relation auf N.

Induktionsprinzip für wohlfundierte Relationen

- Sei
  - R eine wohlfundierte Relation auf einer Menge M,
  - m : A –> M eine Funktion,
  - P eine Eigenschaft vom Elementen aus A.
- **Induktionsprinzip**
  - Zeige für alle a $\in$ A die Eigenschaft P(a) unter der Annahme, dass P(y) gilt für alle y $\in$ A mit m(y) R m(a).
  - Dann gilt P(a) für alle a $\in$ A

Beispiel Induktion über natürlichen Zahlen

1. **Nachfolger-Relation über nat. Zahlen**

   d.h. A = M = N, m(n) = n und xRy, falls y = x + 1

   Dann ist das Induktionsprinzip äquivalent zur vollständigen Induktion, denn für die Gültigkeit von P(n) für alle n $\in$ N muss man zeigen:

   - P(0) ohne Voraussetzungen (da 0 keinen Vorgänger besitzt) und
   - P(k+1) unter der Annahme, dass P(k) gilt (für alle k $\in$ N).

2. **Kleiner-Relation über nat. Zahlen**

   d.h. A = M = N , m(n) = n und xRy gdw. x < y

   Dann ist Folgendes für die Gültigkeit von P(n) für alle n $\in$ N zu zeigen:

   - P(0) ohne Voraussetzungen (da 0 keinen Vorgänger besitzt) und
   - P(k) unter der Annahme, dass P(m) gilt für alle m<k (für alle k $\in$ N).

Beispiel: Schnelle Potenzbildung

- Betrachte nochmals die Exponentialfunktion

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

- Es gilt `exp(x, n) =` x^n.

  Beweis: Nächste Folie

Beispiel: Schnelle Potenzbildung

Es gilt `exp(x, n) = x`^{n}.

Beweis:

- Wähle als wohlfundierte Ordnung die Kleiner-Relation auf natürlichen Zahlen, d.h. M = N und xRy gdw. x < y
- Zeige für alle n ∈ N: P(n) = (∀ `x:int. exp(x, n) = x`^{n})

  Beweis:

  1. Fall `n=0`: `exp(x, 0) = 1 = x`^{0}.

  2. Fall `n>0`: Induktionsannahme: Es gilt `exp(x, m) = x`^{m} für alle `m<n`

  a) n gerade, d.h. n = 2m, n > m >0:

  `exp(x, n) =quadrat(exp(x, n div 2)) =`

  `quadrat(exp(x, m)) =` [Ind.Ann. für m]

  `quadrat(x`^{m}`) = x`^{m} `* x`^{m} `= x`^{2m} `= x`^{n}

  b) n ungerade, d.h. n = 2m+1:

  `exp(x, n) = x *quadrat(exp(x, n div 2))=`

  `x * quadrat(exp(x, m)) =` [Ind.Ann. für m]

  `x *quadrat(x`^{m}`) = x * x`^{m} `* x`^{m} `= x`$^{2m+1}$ `= x`^{n}

3. Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und in normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung der Sonderausdrücke

Umgebung

- Eine **Umgebung** ist eine Liste von **(Wert-) Bindungen** <N = W> wobei N ein Name (Bezeichner) ist und W ein Wert.
- Während einer SML-Sitzung wird eine Umgebung (die aktuelle Umgebung) systematisch aufgebaut. Zu Beginn ist die aktuelle Umgebung leer.
- Bei der Eingabe einer neuen Deklaration

  `val N = A;` oder `val rec N = A;` oder `fun N = A;`

  wird der Ausdruck A in der aktuellen Umgebung ausgewertet .
  - Falls die Auswertung terminiert und den Wert W berechnet, wird eine neue Bindung N=W der aktuellen Umgebung hinzugefügt.
  - Eine frühere Bindung der Form N=W_0 wird dabei entfernt.
  - Ist die Auswertung undefiniert, so entsteht eine Fehlermeldung, bzw. falls Nichtterminierung der Grund ist, wird nichts ausgegeben.
- Bemerkung
  - Kommt ein Name M im Wertteil A einer Deklaration … N = A vor, so wird der Wert von M ermittelt und in W anstelle von M eingefügt, bevor der Eintrag für N in der Umgebung gespeichert wird. So verändert eine spätere Wiederdeklaration von M den Wert von N nicht.

Beispiele Umgebung

- ```
  - val zwei = 2;          U0 = {}
  val zwei = 2 : int       U1 = {zwei=2}
  - val x = zwei*2;
  val x = 4 : int          U2 = {x=4, zwei=2}
  - val y = 5*x;
  val y = 20 : int
  - val x = x+1;
  val x = 21 : int
  - fun qu(x: int) = x*x;
  val qu = fn:int->int     U5 = {qu = fn x => x*x} + U2
  ```

  $U_0 = \{\}$, $U_1 = \{zwei=2\}$, $U_2 = \{x=4, zwei=2\}$, $U_5 = \{qu = fn\ x => x*x\} + U_2$

- Wir definieren

  $$U + \{N_1=W_1, \ldots, N_k=W_k\} = U_1 \cup \{N_1=W_1, \ldots, N_k=W_k\} ,$$

  wobei U_1 aus U durch Entfernen aller eventuell vorhandenen Bindungen von N_1, …, N_k entsteht.
  ```

3.1.1. Arten von Ausdrücken

Nicht alle Ausdrücke haben denselben Zweck:

- *Konstanten*- und *Funktionsdeklarationen* binden Werte (Konstanten oder Funktionen) an Namen; z.B.:

```
val zwei = 2;

fun quadrat(x: int) = x * x;
```

- *Funktionsanwendungen* wenden Funktionen auf Werte an:

  z.B.
  ```
  quadrat(3 + zwei),
  quadrat(3) + quadrat(2)
  ```

Von dem Zweck eines Ausdrucks hängt ab, wie der Ausdruck ausgewertet wird.

Auswertung von Ausdrücken (II)

Wir unterscheiden verschiedene Arten von Ausdrücken:

- *Funktionale Ausdrücke* sind Konstanten und Funktionsanwendungen:
  - *atomar*:

    z.B. `3`, `2`, `true`, `false`, `zwei`,
  - *zusammengesetzt*:

    z.B. `quadrat(3 + 2)`, `3 * 2`, `not false`, `not false <> true`
- *Sonderausdrücke* werden mit den folgenden vordefinierten Konstrukten gebildet:
  - `val` und `fun`, die zur Wertdeklaration dienen
  - `if-then-else`, `case` und das *Pattern Matching* zur Fallunterscheidung
  - die Boole'schen Operatoren `andalso` und `orelse`

[weitere SML-Konstrukte zur Bildung von Sonderausdrücken folgen später]

3.1.2. Die Auswertung von Ausdrücken als Algorithmus

Frage: wie soll die Auswertung von Ausdrücken spezifiziert werden?

Formalisierung notwendig, weil:

- es sich um eine (symbolische) Berechnung handelt,
- die Auswertung von Ausdrücken auf einem Computer durchgeführt werden soll.

$\leadsto$ Auswertung von Ausdrücken als Algorithmus (formal) beschreiben

Anmerkungen:

- also Zurückführung auf schon Bekanntes — nach dem Prinzip des Occam'schen Messers
- Wichtig:
  Durchführung der Auswertung auf einem Computer ist nicht der Hauptgrund für die algorithmische Spezifikation.

  Die Spezifikation dient dem (menschlichen!) Verständnis!

3.1.3. Die Auswertung von Ausdrücken als rekursive Funktion

Skizze des Auswertungsalgorithmus:

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Werte die Teilausdrücke von `A` aus.

2. Wende die Funktion, die sich als Wert des am weitesten links stehenden Teilausdrucks ergibt, auf die Werte an, die sich als Werte aus der Auswertung der restlichen Teilausdrücke ergeben.

$\leadsto$ rekursiver Algorithmus

Die Auswertung von Ausdrücken als rekursive Funktion (II)

Annahmen des Auswertungsalgorithmus:

- Vereinfachung (aber keine prinzipielle Einschränkung):
  Präfixschreibweise für alle Funktionen,
  z.B. `+(1,2)` statt `1 + 2` (keine korrekte SML-Syntax!)
- Gewisse Funktionen stehen zur Verfügung, ohne dass man sie definieren muss:
  → sogenannte Systemfunktionen (Addition, Multiplikation, . . .)

  Andere Funktionen können definiert werden, z.B.:

  ```
  fun quadrat(x) = *(x, x)
  ```

  Dies ist „syntaktischer Zucker“ für:

  ```
  val quadrat = fn(x) => *(x, x)
  ```

  In der aktuellen Umgebung hat der Name `quadrat` als Wert also die Funktion `fn(x) => *(x, x)`.

Die Auswertung von Ausdrücken als rekursive Funktion (III)

Auswertungsalgorithmus:

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Falls `A` atomar ist, dann:
   (a) Falls `A` vordefiniert ist, liefere den vordefinierten Wert von `A`
   (dieser kann auch eine Systemfunktion sein)
   (b) sonst (der Wert `W` von `A` ist durch eine Gleichung in der Umgebung definiert):
   Liefere `W` als Wert von `A`.
   (`W` kann auch eine Funktion der Form $\texttt{fn}(\texttt{F}_1, \ldots, \texttt{F}_\texttt{k})$ `=> R` sein.)

Die Auswertung von Ausdrücken als rekursive Funktion (IV)

2. $\mathtt{A}$ ist zusammengesetzt: d.h. $\mathtt{A}$ hat die Form $\mathtt{B}(\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n})$ mit $\mathtt{n} \geq 0$:

   Werte die Teilausdrücke $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ aus.
   Seien $\mathtt{W}_1, \ldots, \mathtt{W}_\mathtt{n}$ die Werte der Teilausdrücke $\mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$.

   (a) Falls der Wert von $\mathtt{B}$ eine Systemfunktion ist, dann:
   Wende sie auf $(\mathtt{W}_1, \ldots, \mathtt{W}_\mathtt{n})$ an.
   Liefere den dadurch erhaltenen Wert als Wert von $\mathtt{A}$.

   (b) Falls der Wert von $\mathtt{B}$ eine Funktion der Form $\mathtt{fn}(\mathtt{F}_1, \ldots, \mathtt{F}_\mathtt{n}) => \mathtt{R}$ ist, dann:
   Werte $\mathtt{R}$ in der erweiterten Umgebung aus
   (aktuelle Umgebung erweitert um Gleichungen: $\mathtt{F}_1 = \mathtt{W}_1, \ldots, \mathtt{F}_\mathtt{n} = \mathtt{W}_\mathtt{n}$)
   Liefere den dadurch erhaltenen Wert als Wert von $\mathtt{A}$
   (die Umgebung ist nun wieder die ursprüngliche).

Die Auswertung von Ausdrücken als rekursive Funktion (V)

Ist Auswertung von `B` im Fall 2 nicht umständlich und kompliziert?
Betrachten wir die verschiedenen Unterfälle.

- Der SML-Typ `unit` ermöglicht es, dass $\texttt{n} = \texttt{0}$ im Fall 2 vorkommen kann.

  Die folgende Funktionsdeklaration ist möglich:

  ```
  - fun f() = 1;
  val f = fn : unit -> int
  ```

  `f()` ist die Anwendung der Funktion namens `f` auf `()` (unity).

  Für andere funktionale Programmiersprachen kann im Fall 2 des Auswertungsalgorithmus die Einschränkung $\texttt{n} \geq \texttt{1}$ statt $\texttt{n} \geq \texttt{0}$ notwendig sein.

Die Auswertung von Ausdrücken als rekursive Funktion (VI)

- Sei `A` ein zusammengesetzter Ausdruck A mit `abs` (d.h. `B` ist `abs`).

  `abs` ist (nur!) der Name der einstelligen Systemfunktion (vom Typ `int -> int`) zur Berechnung des Absolutbetrags eines Wertes.

  Also muss `B` zunächst „ausgewertet“ werden;
  die sich als Wert ergebende Systemfunktion kann dann angewendet werden.

- Sei `A` der zusammengesetzte Ausdruck `quadrat(2)` (d.h. `B` ist `quadrat`).

  (mit `quadrat` als dem Namen, der in der aktuellen Umgebung die Funktion `fn(x) => *(x, x)` als Wert hat)

  Die Auswertung von `B` liefert also `fn(x) => *(x, x)`.

  Dann wird der Rumpf `*(x, x)` ausgewertet
  (wofür die Umgebung um die Gleichung `x = 2` erweitert wurde).

Die Auswertung von Ausdrücken als rekursive Funktion (VII)

- Sei `A` der Ausdruck `(if n>0 then quadrat else abs)(5)`
  (d.h. `B` = `if n>0 then quadrat else abs` ist selbst zusammengesetzt).

  Hier muß zunächst `B` ausgewertet werden, um die Funktion zu erhalten, die auf `5` angewendet werden soll.

Der vorgestellte Auswertungsalgorithmus definiert eine Funktion, die

- als Eingabeparameter einen Ausdruck und eine Umgebung erhält und
- als Wert den Wert des Ausdrucks liefert.

Die Auswertungsfunktion ist rekursiv! (siehe Fall 2 und 2(b))

3.1.4. Unvollständigkeit des obigen Algorithmus

Der vorgeschlagene Auswertungsalgorithmus ist nicht vollständig:

- Programmfehler werden nicht behandelt.
- Typen werden nicht berücksichtigt.
- Einige Konstrukte von SML können nicht behandelt werden.
- Die Behandlung der Umgebung ist nur unpräzise erläutert.

Unvollständigkeit des obigen Algorithmus (II)

Offenbar können die folgenden konkreten Fehler auftreten:

- In 1(b) tritt ein Fehler auf, wenn `A` ein nichtdeklarierter Name ist, d.h. für den es keine Gleichung in der Umgebung gibt.
- In 2(a) und 2(b) tritt ein Fehler auf, wenn die Anzahl `n` der aktuellen Parameter $A_1, \ldots, A_n$ mit der Stelligkeit des Wertes von `B` nicht übereinstimmt.

  z.B. wenn der folgende Ausdruck ausgewertet werden soll:

  ```
  quadrat(3, 5)
  ```

$\leadsto$ Erkennung und Behandlung solcher Fehler sowie Berücksichtigung von Typen verlangt Ergänzung des obigen Auswertungsalgorithmus !

(wird später behandelt)

3.1.5. Zweckmäßigkeit des obigen Algorithmus

Ist es zulässig bzw. sinnvoll, den Algorithmus als rekursive Funktion zu spezifizieren, wo er doch selbst u.a. zur Auswertung von rekursiven Funktionen dienen soll?

Dreht sich eine solche Spezifikation nicht „im Kreis" ?
(sozusagen wie ein *Perpetuum Mobile* !?!)

Wir bedienen uns hier unserer Intuition von der Auswertung von (rekursiven) Funktionen, um zu verstehen, wie (rekursive) Funktionen ausgewertet werden.

Nicht anders gehen z.B. Linguisten vor, wenn sie in einer Sprache die Grammatik derselben Sprache erläutern.

$\leadsto$ also kein prinzipiell unsinniges Vorgehen !

Zudem können wir uns von der Durchführbarkeit des Auswertungsalgorithmus überzeugen.

3.1.6. Beispiel einer Durchführung des Auswertungsalgorithmus

Seien folgende Deklarationen gegeben:

```
val zwei = 2;
fun quadrat(x) = *(x, x);
```

so dass die Umgebung also aus den beiden folgenden Gleichungen besteht:

```
quadrat = fn(x) => *(x, x)
zwei    = 2
```

Auszuwerten sei der Ausdruck `A`: `quadrat(zwei)`

(im Folgenden beziehen sich die Nummern auf die entsprechenden Fälle des Algorithmus.)

Beispiel . . . (II)

2. `A` ist zusammengesetzt: `B` ist `quadrat`, $\texttt{A}_1$ ist `zwei`, $\texttt{n} = 1$.
   Werte den Teilausdruck `B` aus;
   Nebenrechnung, in der `A` der Ausdruck `quadrat` ist:

   1. `A` ist atomar.

   (b) `A` ist nicht vordefiniert.
   Als Wert von `quadrat` wird aus der Umgebung `fn(x) => *(x,x)` geliefert.

   Ende der Nebenrechnung; Wert von `B` ist `fn(x) => *(x,x)`.
   Werte den Teilausdruck $\texttt{A}_1$ aus;
   Nebenrechnung, in der `A` der Ausdruck `zwei` ist:

   1. `A` ist atomar.

   (b) `A` ist nicht vordefiniert.
   Als Wert von `zwei` wird aus der Umgebung die natürliche Zahl 2 geliefert.

   Ende der Nebenrechnung; Wert von $\texttt{A}_1$ ist 2.

Beispiel . . . (III)

(b) Der Wert von `B` ist keine Systemfunktion, sondern eine Funktion `fn(x) => *(x,x)`.
Die erweiterte Umgebung besteht aus der aktuellen Umgebung und der zusätzlichen Gleichung `x=2`.
Werte `*(x,x)` in dieser erweiterten Umgebung aus;
Nebenrechnung, in der `A` der Ausdruck `*(x,x)` ist.

2. `A` ist zusammengesetzt,
   `B` ist `*`, A_1 ist `x`, A_2 ist `x`, $n = 2$.
   Werte den Teilausdruck `B` aus;
   Nebenrechnung, in der `A` der Ausdruck `*` ist

   1. `A` ist atomar.

   (a) `A` ist vordefiniert.
   Wert von `*` ist die zweistellige Multiplikationsfunktion, also eine Systemfunktion.

   Ende der Nebenrechnung; Wert von `B` ist die Multiplikationsfunktion.

Beispiel . . . (IV)

Werte den Teilausdruck $\texttt{A}_1$ aus;
Nebenrechnung, in der `A` der Ausdruck `x` ist:

1. `A` ist atomar.

(b) `A` ist nicht vordefiniert.
Als Wert von `x` wird aus der (erweiterten) Umgebung die natürliche Zahl 2 geliefert.

Ende der Nebenrechnung; Wert von $\texttt{A}_1$ ist 2
Genauso: Wert von $\texttt{A}_2$ ist 2

(a) Der Wert von `B` ist eine Systemfunktion, nämlich die Multiplikationsfunktion.
Wende sie auf `(2, 2)` an
Der dadurch erhaltene Wert ist `4`.

Ende der Nebenrechnung, Wert von `*(x,x)` ist `4`.
Der dadurch erhaltene Wert von `A` ist also `4`
(die Umgebung ist nun wieder die ursprüngliche, ohne die Gleichung `x`=2).

Anmerkungen zum Auswertungsbeispiel

In Implementierungen wird üblicherweise nicht die textuelle Repräsentation der vom Benutzer definierten Funktionen (wie etwa `fn(x) => *(x,x)` mit dem Namen `quadrat`) verwendet.

Statt dessen wird ein Verweis (Zeiger, Speicheradresse) auf einen „Funktionsdeskriptor“ verwendet, der folgendes enthält:

- die formalen Parameter,
- die Typen und
- den Funktionsrumpf.

Als „Wert“ von `quadrat` wird dann intern die Speicheradresse dieses Funktionsdeskriptors geliefert.

Einen Teil dieser Angaben benutzt das SML-System, um z.B. die Ausgabe `fn: int -> int` für den Typ der Funktion `quadrat` zu ermitteln.

Anmerkungen zum Auswertungsbeispiel (II)

Das Symbol `fn` (oder die Speicheradresse für einen Funktionsdeskriptor) stellt also nur einen Vermerk dar.

- Eine solche Vermerktechnik ist notwendig, weil ein Programm nur Bezeichner (also *Symbole*) bearbeitet.
- Die Begriffe „Funktion" und „Funktionsanwendung" werden erst durch einen Auswertungsalgorithmus verwirklicht – durch symbolische Berechnungen!

Computer führen Berechnungen penibel Schritt für Schritt nach einem vorgebenen präzise festgelegten Verfahren durch:

> $\leadsto$ Fehler oder Ungenauigkeiten im Programm führen damit fast zwangsläufig zu falschen Ergebnissen !

3.1.7. Substitutionsmodell

Der Auswertungsalgorithmus kann auf einer höheren Abstraktionsebene wie folgt erläutert werden:

> Um einen Ausdruck `A` auszuwerten, werden alle Teilausdrücke von `A` durch ihre Definitionen ersetzt, wobei
>
> - die formalen Parameter von Funktionsdefinitionen durch die aktuellen Parameter der Funktionsanwendung ersetzt werden,
> - vordefinierte Konstanten gemäß ihrer Definition durch ihre Werte ersetzt werden und
> - vordefinierte Funktionen gemäß ihrer Definition ersetzt werden.

Diese sehr abstrakte Beschreibung der Auswertung wird *Substitutionsmodell* genannt.

Das Substitutionsmodell ist viel weniger präzise als der vorangehende Algorithmus — insbesondere wird die Reihenfolge der Auswertungsschritte offen gelassen.

3.2. Auswertung in applikativer und in normaler Reihenfolge

- Auswertungsreihenfolge
- applikative Reihenfolge
- normale Reihenfolge

3.2.1. Auswertungsreihenfolge

Die Reihenfolge der Auswertung der Teilausdrücke eines zusammengesetzten Ausdruck wird

- im Auswertungsalgorithmus festgelegt,
- im Substitutionsmodell *nicht* festgelegt.

Frage: Spielt die Auswertungsreihenfolge eine Rolle ?

Betrachten wir z.B. den Ausdruck `quadrat(2 + 1)`.

Der Auswertungsalgorithmus geht wie folgt vor:

- Da es ein zusammengesetzter Asudruck ist, werden die Teilausdrücke `quadrat` ($\leadsto$ Verweis `fn`) und `2 + 1` ($\leadsto$ Wert `3`) ausgewertet.
- Dann erst wird die zu `quadrat` gehörende Funktion auf den aktuellen Parameter `3` angewendet (was zu `3 * 3` führt).

Auswertungsreihenfolge (II)

Der Auswertungsalgorithmus wertet also

- zunächst die aktuellen Parameter (Operanden) einer Funktionsanwendung aus,
- bevor er die Funktion auf die Werte dieser Parameter anwendet.

Achtung:

Der Auswertungsalgorithmus legt aber nicht fest, in welcher Reihenfolge die Teilausdrücke $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_\mathtt{n}$ ausgewertet werden (z.B. von links nach rechts, von rechts nach links, . . .)

Auswertungsreihenfolge (III)

Auswertung des Ausdrucks `quadrat(quadrat(2))`:

mit dem Auswertungsalgorithmus:		*Eine alternative Reihenfolge:*	
0.	`quadrat(quadrat(2))`	0.	`quadrat(quadrat(2))`
1.	`quadrat(2 * 2)`	1.	`quadrat(2) * quadrat(2)`
2.	`quadrat(4)`	2.	`(2 * 2) * (2 * 2)`
3.	`4 * 4`	3.	`4 * 4`
4.	`16`	4.	`16`

$\leadsto$ anderer Ablauf, aber (hier!) dasselbe Ergebnis !

3.2.2. Auswertung in applikativer Reihenfolge

Die Auswertung entsprechend dem vorgestellten Auswertungsalgorithmus — also Parameterauswertung vor Funktionsanwendung — hat die folgenden Namen, die alle dasselbe bezeichnen:

- Auswertung in applikativer Reihenfolge
- Inside-out–Auswertung
- Call-by-value–Auswertung
- strikte Auswertung

Auswertung in applikativer Reihenfolge (II)

Anmerkung:

„Call-by-value“ hat für nicht-funktionale Programmiersprachen eine andere Bedeutung:

- Es bezeichnet Prozedur-/Funktionsaufrufe, bei denen die Werte der Aufrufparameter an die Prozedur/Funktion übergeben werden (als Kopien), aber nicht die Speicheradressen der Originalwerte.

- Als Alternative dazu gibt es „Call-by-reference“, wobei nur die Speicheradressen als Referenzen auf die Werte übergeben werden.

3.2.3. Auswertung in normaler Reihenfolge

Die Auswertung in der anderen Reihenfolge — also Funktionsanwendung vor Parameterauswertung — hat die folgenden Namen:

- Auswertung in normaler Reihenfolge
- Outside-in–Auswertung
- Call-by-name–Auswertung

Anerkung:
In einigen alten Programmiersprachen bezeichnete *call by name* die Weitergabe eines Bezeichners ohne Festlegung eines Wertes noch einer Speicheradresse, also eine rein textuelle Zeichenfolgenersetzung.

Auswertung in normaler Reihenfolge (II)

Wenn alle Funktionsanwendungen terminieren, liefern beide Auswertungsreihenfolgen (applikativ und normal) immer dasselbe Ergebnis.

Bei nichtterminierenden Funktionsanwendungen ist dies nicht immer der Fall.

Beispiel:
Funktion `null`, die für jede ganze Zahl den Wert `0` liefert,
und eine nichtterminierende Funktion `f`:

```
fun null(x : int) = 0;
fun f(x : int) = f(x + 1);
```

Die Auswertung des Ausdrucks `null(f(1))`

- in normaler Reihenfolge liefert den Wert `0`,
- in applikativer Reihenfolge terminiert nicht.

3.2.4. Vorteil der applikativen Reihenfolge gegenüber der normalen Reihenfolge

Am Beispiel des Ausdrucks `quadrat(quadrat(2))` erkennt man folgenden Vorteil der applikativen Reigenfolge:

- Ein Parameter (oder Operand) wird bei einer Auswertung in applikativer Reihenfolge nur einmal ausgewertet,
- bei einer Auswertung in normaler Reihenfolge jedoch unter Umständen mehrmals.

$\leadsto$ Vermeidung redundanter Auswertungen

3.3. Verzögerte Auswertung

Die applikative Auswertungsreihenfolge ist aber nicht immer die vorteilhafteste.

Betrachten wir hierzu noch einmal die einstellige konstante Funktion `null`:

```
fun null(x : int) = 0;
```

3.3.1. Vorteil der normalen Reihenfolge gegenüber der applikativen Reihenfolge

Vergleich der Auswertung des Ausdrucks `null(quadrat(quadrat(quadrat(2))))`:

in applikativer Reihenfolge:

0. `null(quadrat(quadrat(quadrat(2))))`
1. `null(quadrat(quadrat(2 * 2)))`
2. `null(quadrat(quadrat(4)))`
3. `null(quadrat(4 * 4))`
4. `null(quadrat(16))`
5. `null(16 * 16)`
6. `null(256)`
7. `0`

in normaler Reihenfolge:

0. `null(quadrat(quadrat(quadrat(2))))`
1. `0`

Vorteil der normalen Reihenfolge . . . (II)

Die Durchführung der Auswertung der Parameter (oder Operanden) vor der Funktionsanwendung ist nur dann von Vorteil, wenn alle Parameter von der Funktion „verwendet“ werden.

- Die normale Reihenfolge vermeidet die Auswertung von Parametern, wenn diese danach nicht mehr verwendet werden.

- Die applikative Reihenfolge kann also zu überflüssigen Berechnungen führen.

Die verzögerte Auswertung

Die verzögerte Auswertung (*lazy evaluation*, auch *Call-by-need*) bringt Aspekte der Auswertung in applikativer Reihenfolge und in normaler Reihenfolge zusammen.

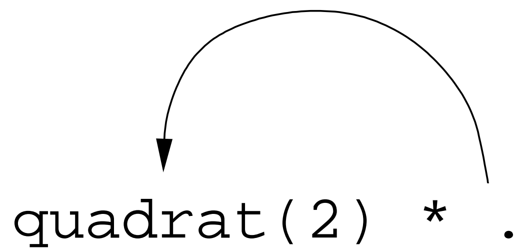
Grundidee der verzögerten Auswertung:

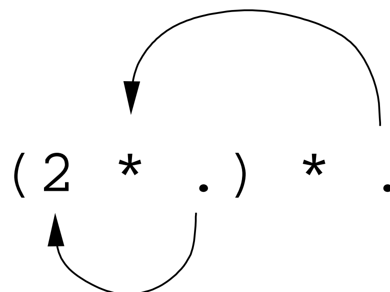
- Wie bei der Auswertung in normaler Reihenfolge führt die verzögerte Auswertung die Funktionsanwendung vor der Auswertung der Parameter (oder Operanden) durch.
- Bei der Funktionsanwendung werden aber alle bis auf ein Vorkommen eines Parameters durch einen Verweis auf ein einziges dieser Vorkommen ersetzt, damit dieser Parameter nicht mehrmals ausgewertet wird.

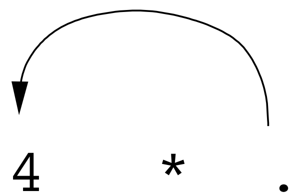
Die verzögerte Auswertung (II)

Auswertung des Ausdrucks `quadrat(quadrat(2))`:

```
quadrat(quadrat(2))
```








```
16
```

Die Verweise („Pfeile“) verhindern, dass mehrfach vorkommende Teilausdrücke mehrfach ausgewertet werden.

Die verzögerte Auswertung (III)

Die verzögerte Auswertung von `null(quadrat(quadrat(quadrat(2))))`:

```
null(quadrat(quadrat(quadrat(2))))

0
```

Analyse:

- Wie die Auswertung in applikativer Reihenfolge vermeidet die verzögerte Auswertung die mehrfache Auswertung von Parametern.
- Wie die Auswertung in normaler Reihenfolge vermeidet die verzögerte Auswertung die Auswertung von Parametern, die zur Auswertung einer Funktionsanwendung nicht notwendig sind.

Die verzögerte Auswertung (IV)

$\Rightarrow$ Die verzögerte Auswertung hat also die Vorteile der beiden Grundansätze zur Auswertung und vermeidet deren Nachteile.

Die verzögerte Auswertung scheint also die beste Auswertungsform zu sein.

Anmerkung:

Das Substitutionsmodell ist nicht zur Formalisierung der verzögerten Auswertung geeignet, weil die verzögerte Auswertung auf einer Datenstruktur (Zeiger bzw. Graph) beruht, die im Substitutionsmodell nicht vorhanden ist.

3.3.3. Auswertungsreihenfolge von SML

SML verwendet die Auswertung in applikativer Reihenfolge, da die die verzögerte Auswertung auch Nachteile hat:

- Die verzögerte Auswertung verlangt eine ziemlich komplizierte — also zeitaufwendige — Verwaltung von Verweisen (Zeigern).
- Die verzögerte Auswertung lässt imperative (oder prozedurale) Befehle wie Schreibbefehle nur schwer zu, weil sie wie die Auswertung in normaler Reihenfolge den Zeitpunkt der Ausführung solcher Befehle schwer vorhersehbar macht.

  (Dies wird bei Einführung der imperativen Befehle von SML näher betrachtet.)
- In manchen Fällen verlangt die verzögerte Auswertung viel mehr Speicherplatz als die Auswertung in applikativer Reihenfolge.

3.4 Auswertung der Sonderausdrücke

1. Wertdeklarationen
2. if-then-else
3. Boole'sche Operatoren andalso und orelse

3.4.1. Wertdeklarationen (val und fun)

- **(Einfache) Wertdeklaration** `val N = A`:

  Es wird die Gleichung `N = A` in die Umgebung eingefügt.
- **Funktionsdeklaration der Form** `val N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt (also wie oben)
- **Rek. Funktionsdeklaration der Form** `val rec N = fn P => A`:

  Es wird die Gleichung `N = fn P => A` in die Umgebung eingefügt. Zusätzlich werden Vorkehrungen getroffen, dass die Umgebung von `A` auch diese Gleichung für `N` enthält.
- **Funktionsdeklaration der Form** `fun N P = A`:

  wie bei der rekursiven Funktionsdeklaration (vorheriger Fall)
- **Bemerkung:**
  - Wird eine Gleichung zu der Umgebung hinzugefügt, so wird sie auf Korrektheit überprüft:. Eine Deklaration wie z.B. `val zwei = zwei` wird dabei abgelehnt. Die verwendeten Namen im Rumpf (hier: zwei) müssen für die Auswertung einer Wertdeklaration bekannt sein.

if-then-else

- Das Grundprinzip der applikativen Reihenfolge, zunächst immer alle Teilausdrücke auszuwerten, kann zur Nichtterminierung bei rekursiv definierten Funktionen führen.
- Daher muss ein Ausdruck

```
if A1 then A2 else A3
```

  als Sonderausdruck mit eigenem (nichtstrikten) Auswertungsmechanismus behandelt werden:
  - Werte von den drei Teilausdrücken `A1, A2, A3` zuerst nur `A1` aus.
  - Hat A1 den Wert true, dann (und nur dann) wird `A2` ausgewertet (und `A3` wird nicht ausgewertet). Der Wert von `A2` wird als Wert des gesamten Ausdrucks geliefert.
  - Hat `A1` den Wert false, dann (und nur dann) wird `A3` ausgewertet (und `A2` wird nicht ausgewertet). Der Wert von `A3` wird als Wert des gesamten Ausdrucks geliefert.

Die Boole'schen Operatoren andalso und orelse

- Ebenso werden die Boole'schen Operatoren andor und orelse **sequentiell nichtstrikt** ausgewertet.
- **Sequentielle Konjunktion (andalso, sequ. $\wedge$)**
  - Zunächst wird nur der erste Teilausdruck `A1` ausgewertet. Ist der Wert von `A1 true`, so wird auch `A2` ausgewertet und dessen Wert als Wert des Ausdrucks `A1` $\wedge$ `A2` geliefert.
  - Ist der Wert von `A1 false`, so wird `false` als Wert des Ausdrucks `A1` $\wedge$ `A2` geliefert (und `A2` wird nicht ausgewertet).
- **Sequentielle Disjunktion (orelse, sequ. $\vee$)**
  - Zunächst wird nur der erste Teilausdruck `A1` ausgewertet. Ist der Wert von `A1 false`, so wird auch `A2` ausgewertet und dessen Wert als Wert des Ausdrucks `A1` $\vee$ `A2` geliefert.
  - Ist der Wert von `A1 true`, so wird `true` als Wert des Ausdrucks `A1` $\vee$ `A2` geliefert (und `A2` wird nicht ausgewertet).

Die Boole'schen Operatoren andalso und orelse

- Wie kann man herausfinden, wie eine Programmiersprache die Boole'schen Operationen auswertet?
- In SML reicht dazu der folgende kleine Test (hier für die Konjunktion):

```
fun roedelBool(n) : bool =
          roedelBool(n + 1);
false andalso roedelBool(0);
```

- Terminiert dieser Ausdruck, dann wertet die Programmiersprache die Konjunktion nichtstrikt aus.

Zusammenfassung

- Eine Relation $R \subseteq M \times M$ heißt **wohlfundiert**, wenn jede absteigende Kette endlich ist. Wohlfundierte Relationen benützt man zum Beweis der Terminierung mit Abstiegsfunktionen; außerdem erhält man ein sehr mächtiges Induktionsprinzip.
- Ein SML-Programm wird in einer **aktuellen Umgebung** ausgewertet, die aus einer Menge von **Wertbindungen** besteht.
- Der **Auswertungsalgorithmus von SML** für einen Ausdruck A geht wie folgt vor:
  - Werte die Teilausdrücke von A aus.
  - Wende die Funktion, die sich als Wert des am weitesten links stehenden Teilausdrucks ergibt, auf die Werte an, die sich als Werte aus der Auswertung der restlichen Teilausdrücke ergeben.
- Im allgemeineren **Substitutionsmodell** werden zur Auswertung eines Ausdrucks A alle Teilausdrücke von A durch ihre Definitionen ersetzt; es wird aber keine Auswertungsreihenfolge (Strategie) festgelegt.
- Mögliche Auswertungsstrategien sind
  - applikative Reihenfolge (Call-by-Value, Inside-Out) wie in SML
  - „Normale" Reihenfolge (Call-by-Name, Outside-In)
  - Verzögerte Auswertung (lazy evaluation, Call-by-need)