5 Operatorwerte

5.1 Operatoren als Parameterwerte

5.1.1 Grundprinzip

- Wenn der Typ eines Parameters ein Operatortyp (d. h. eine Operatordeklaration) ist (vgl. § 3.7.1), kann als zugehöriger Operand – egal ob der Parameter anonym und/oder implizit ist oder nicht – explizit ein Ausdruck angegeben werden, der als Wert einen passenden Operator liefert, d. h. einen Operator, durch den der im Parametertyp deklarierte Operator näherungsweise ersetzt werden kann (vgl. § 3.8.4).

- Konkret kann z. B. eine Konstante übergeben werden, die zuvor mit einem passenden Operator initialisiert wurde.

  Dabei gilt für Konstantendeklarationen ergänzend zu den Regeln in § 2.4:
  Wenn der Typ der Konstante ein Operatortyp ist und die Konstante keine explizite Initialisierung besitzt, wird sie nicht mit einem neuen synthetischen Wert initialisiert, sondern mit dem im Operatortyp deklarierten Operator (der anschließend auch wie gewohnt verwendet werden kann), zum Beispiel:

  ```
  square: (x:int) "^2" -> (int = x * x)
  ```

  Der Wert der Konstanten `square` ist somit der hier definierte Operator $\bullet^2$.

- Anders als in § 3.10.2, ist bei der expliziten Übergabe eines Operators an einen Parameter nur näherungsweise Ersetzbarkeit nötig, d. h. alle Namen in der Hauptsignatur des Operators und des zugehörigen Parameters werden gemäß § 3.8.4 ignoriert, damit auch ein Operator mit anderen Namen, die sich eventuell auch an anderen Stellen der Signatur befinden, übergeben werden kann, zum Beispiel (vgl. § 3.7.2):

```
print values of (f (int) -> (int)) from (x1:int) to (x2:int)
-> (int =
  for x = x1 .. x2 do
    print only x;
    print only '-'; print only '>';
    print f x
  end
);

print values of square from 1 to 10
```

Obwohl die Typen der Konstanten `square` (d. h. `(int) "²" -> (int)`) und des Parameters `f` (d. h. `f (int) -> (int)`) unterschiedliche Namen an unterschiedlichen Stellen ihrer Hauptsignatur besitzen, kann der Operator `square` an den Parameter übergeben werden, weil der Parameter näherungsweise durch den Operator ersetzt werden kann.

- Analog zu § 3.10.2, kann der übergebene Operator auch allgemeiner als der Parameter sein, zum Beispiel (vgl. § 3.10.3):

```
generic square:
[(T:type)] (x:T) "^2" [(+ (T) "*" (T) -> (T))] -> (T = x * x);

print values of generic square from 1 to 10
```

Der Operator `generic square` kann an den Parameter `f` übergeben werden, weil jede korrekte Anwendung des Parameters auch eine korrekte Anwendung des Operators mit dem gleichen Typ ist, sofern man die Namen des Parameters und des Operators ignoriert (und sofern es einen Multiplikationsoperator für den jeweiligen Typ `T` gibt).

- Der zu übergebende Operator kann auch – vergleichbar mit Lambda-Ausdrücken in anderen Sprachen – erst an seiner Verwendungsstelle deklariert werden.
  Aber weil eine Operatordeklaration nicht den deklarierten Operator, sondern seinen Typ liefert, ist trotzdem eine Konstantendeklaration erforderlich, die aber auch anonym sein kann, zum Beispiel:

```
print values of : (x:int) "^2" -> (int = x * x) from 1 to 10
```

- Um den „syntaktischen Ballast“ noch weiter zu reduzieren, können einerseits die Namen des zu übergebenden Operators weggelassen werden (weil sie ohnehin bedeutungslos sind) und andererseits auch seine Parameter- und Resultattypen (sofern sie vom Übersetzer deduziert werden können, was meist der Fall ist), zum Beispiel:

```
print values of : (x:) -> (= x * x) from 1 to 10
```

  Der Doppelpunkt nach dem Namen eines Parameters und das Gleichheitszeichen vor der Implementierung des Operators sind aber notwendig, weil der Parametername sonst als Typ eines anonymen Parameters und der Implementierungsausdruck als Resultattyp des Operators interpretiert werden könnte.

5.1.2 Typische Beispiele aus der funktionalen Programmierung

Map

- `map xs using f` wendet auf alle Elemente der Liste `xs` (vgl. § 4.3.4) die Funktion bzw. den Operator `f` an und liefert als Resultat eine neue Liste mit den Resultatwerten der Funktionsaufrufe:

```
[(X:type) (Y:type)]
map (xs:X*) using (f: f (X) -> (Y)) -> (Y* =
  if xs then f xs.head -> map xs.tail using f end
)
```

  Damit die Funktion `f` an den rekursiven Aufruf von `map` weitergegeben werden kann, darf der entsprechende Parameter hier nicht anonym sein.

- Zum Beispiel:

```
xs := 1 -> 2 -> 3 -> 4 -> 5 ->;
print map xs using square;                  $$ 1->4->9->16->25->
print map (xs) using : (x:) -> (= 2*x)      $$ 2->4->6->8->10->
```

  (Ohne die Klammern um `xs` könnte die letzte Zeile auch als Deklaration einer Konstanten mit den Namen `print map xs using` interpretiert werden.)

Filter

- `filter xs using f` wendet auf alle Elemente der Liste `xs` die Funktion `f` an und liefert als Resultat eine neue Liste, die alle Elemente von `xs` enthält, für die der Funktionsaufruf nicht nil geliefert hat:

```
[(X:type) (Y:type)]
filter (xs:X*) using (f: f (X) -> (Y)) -> (X* =
  if xs then
    h := xs.head;
    r := f h;
    t := filter xs.tail using f;
    if r then h -> t else t end
  end
)
```

  Damit die Reihenfolge der Aufrufe von `f` der Reihenfolge der Listenelemente entspricht, muss `f h` vor dem rekursiven Aufruf von `filter` ausgeführt werden.

- Zum Beispiel:

```
print filter (xs) using : (x:) -> (= x -:- 2 = 0)
                                        $$ 2->4->
```

Fold

- `fold x and xs using f` verknüpft den Wert `x` und die Elemente der Liste `xs` linksassoziativ durch sukzessive Aufrufe der binären Funktion `f` und liefert das Ergebnis des letzten Aufrufs. Wenn die Liste `xs` leer ist, wird direkt der Wert `x` geliefert:

```
[(X:type)]
fold (x:X) and (xs:X*) using (f: f (X) (X) -> (X)) -> (X =
  if xs then fold f x xs.head and xs.tail using f else x end
)
```

- Entsprechend verknüpft `fold xs and x using f` die Elemente der Liste `xs` und den Wert `x` rechtsassoziativ durch sukzessive Aufrufe von `f` und liefert das Ergebnis des letzten Aufrufs. Wenn die Liste `xs` leer ist, wird wiederum direkt der Wert `x` geliefert:

```
[(X:type)]
fold (xs:X*) and (x:X) using (f: f (X) (X) -> (X)) -> (X =
  if xs then f xs.head fold xs.tail and x using f else x end
)
```

- Zum Beispiel:

```
print fold 0 and xs using : (x:) (y:) -> (= x + y);      $$ 15
print fold xs and 1 using : (x:) (y:) -> (= x * y)       $$ 120
```

- Unterschied zwischen Links- und Rechtsfaltung:

```
print fold 0 and xs using : (x:) (y:) -> (= x - y);      $$ -15
print fold xs and 0 using : (x:) (y:) -> (= x - y)       $$ 3
```

  Achtung: Wenn es einen Präfixoperator `-•` für Vorzeichenwechsel gäbe, könnte `x - y` auch als rekursive Anwendung des an `fold` übergebenen Operators auf die Operanden `x` und `-y` interpretiert werden. Dann müsste man statt `x - y` z. B. `x + -y` schreiben.

Anmerkung

- Da Listen hier, wie in der funktionalen Programmierung üblich, rekursiv definiert sind – eine Liste ist entweder leer, oder sie besteht aus einem ersten Element `head` und einer Restliste `tail` –, können die Operatoren `map`, `filter` und `fold` analog rekursiv implementiert werden, was wiederum typisch für funktionale Programmierung ist:
  Wenn die Liste leer ist, endet die Rekursion jeweils, andernfalls wird der Operator rekursiv für die Restliste aufgerufen. Die „eigentliche“ Operation wird in jedem Rekursionsschritt nur für das erste Element der aktuellen Liste ausgeführt.

- Bei sehr langen Listen besteht durch die Rekursion jedoch die Gefahr eines Stapelüberlaufs, weil der MOSTflexiPL-Übersetzer – im Gegensatz zu den Übersetzern vieler funktionaler Programmiersprachen – typische Rekursionsmuster nicht automatisch in äquivalenten iterativen Code transformiert.

5.2 Operatoren als Resultatwerte

- Wenn der Resultattyp einer Operatordeklaration ein Operatortyp ist, muss die Implementierung des Operators einen passenden Operator liefern, d. h. wiederum einen Operator, durch den der im Resultattyp deklarierte Operator näherungsweise ersetzt werden kann.

- Dieser kann z. B. als lokaler Operator, wiederum kombiniert mit einer anonymen Konstantendeklaration, definiert werden, zum Beispiel:

```
add (y:int) -> ((int) -> (int) =
  : (x:int) -> (int = x + y)
)
```

  Für einen `int`-Wert `y` liefert `add y` einen Operator, der einen `int`-Wert `x` auf den `int`-Wert `x + y` abbildet.

- Mögliche Verwendung (vgl. § 4.3.4 und § 5.1.2):

```
is := 1 -> 2 -> 3 -> 4 -> 5 ->;
print map is using add 10        $$ 11->12->13->14->15->
```

- Damit `add` auch für andere Typen als `int` verwendet werden kann, für die es einen passenden Additionsoperator gibt (beispielsweise `rat`, vgl. die Aufgabe zu rationalen Zahlen), kann es wie folgt verallgemeinert werden:

```
[(T:type) (+ (T) "+" (T) -> (T))]
add (y:T) -> ((T) -> (T) =
  : (x:T) -> (T = x + y)
)
```

- Mögliche Verwendung dann:

```
rs := 1/2 -> 2/3 -> 3/4 ->;
print map rs using add 1/8      $$ 5/8->19/24->7/8->
```

- Damit `add` noch allgemeiner verwendet werden kann (z. B. wenn es einen Additionsoperator gibt, der einen `rat`- und einen `int`-Wert addiert und als Ergebnis einen `rat`-Wert liefert), kann es noch generischer definiert werden:

```
[(X:type) (Y:type) (Z:type) (+ (X) "+" (Y) -> (Z))]
add (y:Y) -> ((X) -> (Z) =
  : (x:X) -> (Z = x + y)
)
```

- Mögliche Verwendung dann:

```
print map rs using add 5        $$ 11/2->17/3->23/4->
```

- Definition von `mul` analog zu `add`:

```
[(X:type) (Y:type) (Z:type) (+ (X) "*" (Y) -> (Z))]
mul (y:Y) -> (=
  : (x:X) -> (= x * y)
)
```

Der Resultattyp von `mul` ist der Typ des lokal definierten Operators (nämlich `(X) -> (Z)`), der vom Übersetzer, genauso wie der Resultattyp dieses lokalen Operators (`Z`), automatisch ermittelt wird.
Auch für `mul` ist es sinnvoll, dass es maximal generisch mit drei verschiedenen Typparametern definiert ist, damit es z. B. für Matrizen verwendbar ist, wo die Multiplikation einer $L \times M$- mit einer $M \times N$-Matrix eine $L \times N$-Matrix liefert (vgl. die Aufgabe zu Matrizen).

- Komposition von Funktionen:

```
[(X:type) (Y:type) (Z:type)]
(f (X) -> (Y)) before (g (Y) -> (Z)) -> (=
  : (x:X) -> (= g f x)
)
```

Für eine Funktion `f`, die einen Wert des Typs `X` auf einen Wert des Typs `Y` abbildet, und eine Funktion `g`, die einen Wert des Typs `Y` auf einen Wert des Typs `Z` abbildet, liefert `f before g` eine Funktion, die auf einen Wert des Typs `X` zunächst die Funktion `f` und auf den Ergebniswert dann die Funktion `g` anwendet.

- Mögliche Verwendung:

```
print map is using mul 2 before add 1;      $$ 3->5->7->9->11->
print map rs using add 1/8 before mul 2/3   $$ 5/12->19/36->7/12->
```

- Damit beliebig viele Funktionen miteinander verkettet werden können, kann •`before`• entweder links- oder rechtsassoziativ definiert werden:

```
bind (: (int) -> (int)) before (: (int) -> (int)) left end
```

- Mögliche Verwendung:

```
print map is using mul 2 before add 1 before mul 3
                                                $$ 9->15->21->27->33->
```

5.3 Operatoren als Variablenwerte

5.3.1 Grundprinzip

- Wenn der Inhaltstyp einer Variablen ein Operatortyp ist, kann die Variable einen passenden Operator enthalten, d. h. wiederum einen Operator, durch den der im Inhaltstyp deklarierte Operator näherungsweise ersetzt werden kann, zum Beispiel:

```
logger : (s:char*) -> (bool) ?
```

- Die Variable `logger` kann beliebige Operatoren mit Parametertyp `char*` (vgl. § 4.3.4 und § 4.5) und Resultattyp `bool` enthalten.
  (Der Parametername `s` ist notwendig, weil `(char*) -> (bool)` sonst auch als Anwendung des in § 4.3.2 definierten Operators •`->`• interpretiert werden kann, dessen Operanden beide mit redundanten Klammern umgeben sind.
  Alternativ könnte man dem Operator `(char*) -> (bool)` einen beliebigen Namen geben, z. B. `log (char*) -> (bool)`).

- Zum Beispiel:

```
simple logger: (msg:char*) -> (bool =
  print msg
);

colored logger: (msg:char*) -> (bool =
  esc := char 27;
  print only esc -> "[31m";
  print only msg;
  print esc -> "[39m"
);

logger =! if ..... then simple logger else colored logger end
```

- Der Operator `simple logger` gibt die Zeichenkette `msg` normal aus, während der Operator `colored logger` durch Ausgabe von Steuerzeichen dafür sorgt, dass sie rot ausgegeben wird.
  (Die Zeichenfolge `esc [ 3 1 m` schaltet die Vordergrundfarbe einer ANSI-konformen Konsole auf Rot, während `esc [ 3 9 m` wieder auf die Standardfarbe zurückschaltet. `esc` ist das Escape-Zeichen mit Dezimalwert 27.)
  Abhängig von irgendeiner Laufzeitbedingung wird in der Variablen `logger` dann entweder der Operator `simple logger` oder der Operator `colored logger` gespeichert.

- Der in einer Variablen enthaltene Operator kann zum Beispiel als Parameterwert an einen anderen Operator übergeben werden, zum Beispiel:

```
(f (char*) -> (bool)) (s:char*) -> (bool = f s);
?logger "Log-Ausgabe"
```

  Der in der ersten Zeile definierte Operator, dessen Signatur lediglich aus zwei Parametern besteht, wendet den Operator, der an den ersten Parameter `f` übergeben wird, auf die Zeichenkette an, die an den zweiten Parameter `s` übergeben wird. Dementsprechend wird in der zweiten Zeile der Operator, der momentan in der Variablen `logger` gespeichert ist, auf die Zeichenkette `"Log-Ausgabe"` angewandt.

- Der zuvor definierte Operator kann auch generisch definiert werden:

```
[(X:type) (Y:type)] (f (X) -> (Y)) (x:X) -> (Y = f x);
?logger "Log-Ausgabe"
```

5.3.2 Weitere Möglichkeiten

- Da Operatoren in Variablen gespeichert werden können, können sie z. B. auch in Attributen offener Typen (vgl. § 4.3.2) oder in ähnlichen Datenstrukturen (vgl. § 4.4) gespeichert werden.

- Zum Beispiel:

```
[(T:type)] observer => (T? -> (var:T?) -> (bool));

[(T:type)] observe (var:T?) with (f: (T?) -> (bool)) -> (bool =
  var@observer =! f;
  true
)
```

- `observer` definiert ein generisches Attribut (vgl. § 4.3.4), das jeder Variablen mit irgendeinem Typ `T?` einen Operator mit Typ `(var:T?) -> (bool)` zuordnet.
(Auch hier ist der Parametername `var` wieder notwendig, damit der zweite Pfeil im Ausdruck `T? -> (var:T?) -> (bool)` eindeutig eine Operatordeklaration und damit einen Operatortyp bezeichnet und nicht wieder als Anwendung des in § 4.3.2 definierten Pfeiloperators interpretiert werden kann.
Der erste Pfeil in diesem Ausdruck bezeichnet jedoch genau diesen Pfeiloperator.)

- `observe var with f` speichert dann den Operator `f` im Attribut `observer` der Variablen `var`, zum Beispiel:

```
x : int?;

observe (x) with
: (var:int?) -> (bool = print "observer for variable x")
```

- Damit ein so zugeordneter „Beobachter“ einer Variablen bei jeder nachfolgenden Zuweisung an die Variable aufgerufen wird, kann der vordefinierte Zuweisungsoperator •=!• durch einen benutzerdefinierten Operator mit derselben Signatur überschrieben (eigentlich verdeckt) werden (siehe auch § 3.6.1):

```
[(T:type)] (var:T?) "=!!" (val:T) -> (T = var =! val);

["["(T:type)"]"] (var:T?) "=!" (val:T) -> (T =
    var =!! val;
    var.observer var;
    val
)
```

Damit der benutzerdefinierte Operator den dadurch verdeckten vordefinierten Zuweisungsoperator aufrufen kann, muss für diesen zuvor noch ein „Alias“ mit anderer Signatur (hier •=!!•) definiert werden.

- Damit wird der Operator, der der Variablen `x` zuvor mittels `observe x with ...` zugeordnet wurde, bei jeder nachfolgenden Zuweisung `x =! ...` aufgerufen.

- Damit ein Beobachter auch für mehrere verschiedene Variablen verwendet werden kann, erhält er die jeweilige Variable als Parameterwert, nachdem ihr neuer Wert an sie zugewiesen wurde.

- Wenn einer Variablen `var` kein Beobachter zugeordnet wurde, liefert `var.observer` nil, d. h. einen nil-Operator, der keine Implementierung besitzt und dessen Aufruf somit wirkungslos ist.