4 Statische Operatoren

4.1 Grundprinzip

- Wenn bei der Deklaration eines Operators ein Doppelpfeil => anstelle eines einfachen Pfeils -> verwendet wird, handelt es sich um einen *statischen* Operator. (Operatoren, die mit einem einfachen Pfeil definiert sind, heißen zur Unterscheidung auch *dynamische* Operatoren.)

- Ein statischer Operator hat ein „Gedächtnis“, in dem er sich alle bisherigen Aufrufe und die zugehörigen Resultatwerte „merkt“.

- Wenn er erneut mit der gleichen Syntax und den gleichen Parameterwerten wie früher aufgerufen wird, liefert er direkt den damaligen Resultatwert aus dem Gedächtnis, d. h. in so einem Fall wird die Implementierung des Operators nicht erneut ausgewertet.

- Das kann prinzipiell dazu verwendet werden, Operatoren automatisch tabellengestützt zu optimieren (dynamic programming). Allerdings sollte die Anzahl verschiedenartiger Aufrufe dann nicht zu groß sein.

- Wesentlich wichtiger ist jedoch die Möglichkeit, damit benutzerdefinierte Datenstrukturen zu implementieren.

4.2 Operatoren ohne explizite Implementierung

- Wenn bei der Deklaration eines (statischen oder dynamischen) Operators keine explizite Implementierung angegeben wird, besitzt der Operator als Implementierung implizit einen Ausdruck, der bei jeder Auswertung einen neuen (und damit eindeutigen) synthetischen Wert liefert (so wie der Operator `uniq` in § 3.7.3).

- Da die Implementierung eines statischen Operators bei einem Aufruf mit gleicher Syntax und gleichen Parameterwerten wie früher aber nicht ausgewertet wird, erhält man in diesem Fall denselben synthetischen Wert wie zuvor.

4.3 Offene Typen

4.3.1 Erste Idee

```
Point : type;

(p:Point) "@" x => (int?);
(p:Point) "@" y => (int?);

p1 : Point; p1@x =! 3; p1@y =! 4;
p2 : Point; p2@x =! 5; p2@y =! 6;

(p:Point) "." x -> (int = ?p@x);
(p:Point) "." y -> (int = ?p@y);

print only p1.x; print only ' '; print p1.y;     $$ 3 4
print only p2.x; print only ' '; print p2.y     $$ 5 6
```

Erläuterungen

- `Point` ist eine Konstante des Metatyps `type` mit einem neuen synthetischen Wert und stellt deshalb einen neuen eindeutigen Typ dar.
- Für einen Wert `p` dieses Typs `Point` liefert `p@x` bzw. `p@y` jeweils einen synthetischen Wert des Typs `int?`, d. h. eine Variable mit Inhaltstyp `int`.
- Weil es sich um statische Operatoren handelt, erhält man beim ersten Aufruf für einen bestimmten Punkt `p` jeweils eine neue Variable, bei einem späteren Aufruf mit dem gleichen Punkt `p` jedoch dieselbe Variable wie zuvor.
- `p1` und `p2` sind Konstanten des Typs `Point` mit einem jeweils neuen synthetischen Wert und stellen deshalb eindeutige Objekte des Typs `Point` dar.
- Deshalb liefert jeder der Ausdrücke `p1@x`, `p1@y`, `p2@x` und `p2@y` eine andere, aber bei jedem Aufruf die gleiche Variable, die dementsprechend zur Speicherung der jeweiligen Koordinate des jeweiligen Punkts verwendet werden kann.
- Der Ausdruck `p.x` bzw. `p.y` stellt lediglich eine Abkürzung für `?p@x` bzw. `?p@y` dar.
- Weil sich der Inhalt der Variablen `p@x` und `p@y` zwischen Aufrufen von `p.x` und `p.y` ändern kann, müssen diese Operatoren dynamisch sein.

4.3.2 Verallgemeinerung und Verbesserung

Generische Definitionen

```
(U:type) "->" (V:type) => (type);

[(U:type) (V:type)]
(u:U) "@" (a:U->V) => (V? = u /\ a /\ v:V?);

[(U:type) (V:type)]
(u:U) "." (a:U->V) -> (V = ?u@a);

excl
  U : type; u1 : U; a1 : U -> U; u2 : U; a2 : U -> U; v : int?;
  (u1@a1) <-> (?v); u1 <-> v; a1 <-> v;
  (u2.a2) <-> (?v); u2 <-> v; a2 <-> v
end
```

Konkrete Verwendung

```
Point : type;
x : Point -> int;
y : Point -> int;

p1 : Point; p1@x =! 3; p1@y =! 4;
p2 : Point; p2@x =! 5; p2@y =! 6;

print only p1.x; print only ' '; print p1.y;     $$ 3 4
print only p2.x; print only ' '; print p2.y      $$ 5 6
```

Erläuterungen

- Für zwei beliebige Typen `U` und `V` liefert `U->V` jeweils einen eindeutigen Typ, der zur Repräsentation von *Attributen* des Typs `U` mit *Zieltyp* `V` dient.
- Für ein Objekt `u` eines beliebigen Typs `U` und ein Attribut `a` dieses Typs mit Zieltyp `V` liefert `u@a` normalerweise (wenn `u` und `a` nicht nil sind) jeweils eine eindeutige Variable `v` mit Inhaltstyp `V`, die zur Speicherung des Werts des Attributs `a` des Objekts `u` dient.
- Ausnahme: Wenn `u` oder `a` nil ist, liefert `u@a` ebenfalls nil (d. h. eine nil-Variable), sodass Zuweisungen an `u@a` dann wirkungslos sind und `?u@a` wiederum nil liefert (vgl. § 2.7; die Konjunktion `•/\•`, deren rechter Operand nur bei Bedarf ausgewertet wird, wird auf einem Aufgabenblatt definiert).
- `u.a` ist wiederum nur eine Abkürzung für `?u@a`.
- Die Operatoren `•@•` und `•.•` sollen die gleichen Bindungseigenschaften wie die vordefinierte Variablenabfrage besitzen.
- In der konkreten Verwendung sind `x` und `y` Konstanten des Typs `Point->int` mit jeweils eindeutigen Werten, die somit zwei verschiedene Attribute des Typs `Point` mit Zieltyp `int` darstellen.
- Damit haben Ausdrücke wie `p1@x`, `p2@y`, `p1.y` etc. dieselbe Bedeutung wie zuvor.

4.3.3 Generischer Konstruktor und Attributänderungsoperator

Generische Definitionen

```
$$ Neues Objekt des offenen Typs U erzeugen
$$ und v1, v2 ... als Werte der Attribute a1, a2 ... speichern.
(U:type) "(" [(V1:type)] (a1:U->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U->V2) "=" (v2:V2) } ")" -> (U =
  u : U;
  u@a1 =! v1;
  { u@a2 =! v2 };
  u
);

$$ v1, v2 ... als neue Werte der Attribute a1, a2 ...
$$ des Objekts u speichern.
[(U:type)] (u:U) "(" [(V1:type)] (a1:U->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U->V2) "=" (v2:V2) } ")" -> (U =
  u@a1 =! v1;
  { u@a2 =! v2 };
  u
)
```

Konkrete Verwendung

```
$$ Punkt p mit Koordinaten 1 und 2 erzeugen.
p := Point(x = 1, y = 2);

$$ Koordinaten von p auf 5 und 6 setzen.
p(y = 6, x = 5)
```

4.3.4 Generische offene Typen

Beispiel: Listen

```
$$ Generischer Typ T*
$$ zur Repräsentation von Listen mit Elementtyp T.
(T:type) "*" => (type);

$$ * soll stärker binden als ->,
$$ damit T -> T* als T -> (T*) interpretiert wird.
excl (int -> int)* end;

$$ Generische Attribute head und tail von T*.
[(T:type)] head => (T* -> T);
[(T:type)] tail => (T* -> T*);

$$ Liste mit erstem Element h und optionaler Restliste t.
[(T:type)] (h:T) "->" [(t:T*)] -> (T* =
  T*(head = h, tail = t)
);
```

```
$$ -> soll stärker binden als Konstantendeklaration,
$$ damit ls := 1 -> als ls := (1 ->) interpretiert wird.
excl (x := 1) -> end;

$$ Länge der Liste ls.
[(T:type)] "#" (ls:T*) -> (int =
  p : T*?;
  p =! ls;
  while ?p do p =! ?p.tail end
)
```

Exemplarische Verwendung

```
$$ Listen mit unterschiedlichen Elementtypen erzeugen
$$ und ihre Länge ausgeben.

ls := 1 -> 2 -> 3 -> 4 -> 5 ->;
print #ls;                          $$ 5

ls2 := 'a' -> 'b' -> 'c' ->;
print #ls2                          $$ 3
```

Erläuterungen

- Da der Typ der Attribute `head` und `tail` jeweils von ihrem Typparameter `T` abhängt, können diese Attribute nicht als Konstanten definiert werden, sondern stattdessen als statische Operatoren.

- Bei typischen Verwendungen von `head` und `tail` wird die Belegung von `T` und damit der konkrete Typ des Attributs jeweils aus dem Verwendungskontext ermittelt (vgl. § 3.7.3).

Ausgabe von Listen

```
$$ Elemente der Liste ls ausgeben,
$$ sofern es einen Ausgabeoperator für ihren Elementtyp T gibt.
[(T:type)] print <o>[only] (ls:T*)
                  [(+ print only (T) -> (bool))] -> (bool =
  p : T*?;
  p =! ls;
  while ?p do
    print only ?p.head;
    print only '-'; print only '>';
    p =! ?p.tail
  end;
  <o>[true | print 1:0]
)
```

Exemplarische Verwendung

```
print ls2;                  $$ a->b->c->

ls3 := ls2 -> ls2 ->;
print ls3                   $$ a->b->c->->a->b->c->
```

4.3.5 Anmerkungen

- Da zu einem einmal definierten Typ jederzeit – auch an anderen Stellen eines Programms oder sogar in anderen Modulen – weitere Attribute hinzugefügt werden können, sind diese Typen offen für nachträgliche Erweiterungen und werden deshalb als *offene Typen* bezeichnet.

- Außerdem müssen nicht alle Objekte eines solchen Typs jeweils Werte für alle Attribute des Typs besitzen. Nicht vorhandene Attributwerte belegen dann auch keinen Speicherplatz, und bei ihrer Abfrage erhält man jeweils nil.

- Damit eignen sich offene Typen auch sehr gut (insbesondere wesentlich besser als `union`-Typen in C) zur flexiblen Speicherung varianter Datenstrukturen.

4.4 Reihenartige Typen

Generische Definitionen

```
$$ Generischer Typ T []
$$ zur Repräsentation von Reihen (arrays) mit Elementtyp T.
(type) "[" "]" => (type);

$$ Zugriff auf das i-te Element der Reihe a.
[(T:type)] (a:T[]) "@" (i:int) => (T? = a /\ i /\ (v:T?));
[(T:type)] (a:T[]) "." (i:int) -> (T = ?a@i)
```

Konkrete Verwendung

```
letters : char [];
letters@1 =! 'a';
letters@26 =! 'z';

print letters.1;                $$ a
print letters.2;                $$ (nichts)
print letters.26                $$ z
```

4.5 Zeichenketten

- In MOSTflexiPL gibt es bewusst keinen vordefinierten Typ für Zeichenketten, weil man sich selbst leicht (typischerweise in einer Bibliothek) einen geeigneten Typ definieren kann.
  Konkret kann z. B. der Listentyp `char*` (vgl. § 4.3.4) oder ein reihenartiger Typ (vgl. § 4.4) verwendet werden.
- Trotzdem gibt es vordefinierte *Zeichenkettenliterale*, die aus beliebig vielen Unicode-Zeichen innerhalb von (doppelten) Anführungszeichen bestehen.
  Wie bei Namen von Konstanten (vgl. § 2.4) und Operatoren, muss ein solches Anführungszeichen verdoppelt werden, um es in ein Zeichenkettenliteral zu integrieren.
  (Der einzige Unterschied zu solchen Namen ist, dass Zeichenkettenliterale auch Zwischenraum enthalten und leer sein können.)
- Um ein solches Literal verwenden zu können, muss es im aktuellen Kontext einen Operator des Typs

  ```
  str { (c:char) } -> (S)
  ```

  mit einem beliebigen Resultattyp `S` geben, der implizit an das Literal übergeben wird (vgl. § 3.11) und von diesem mit allen Zeichen des Literals aufgerufen wird.
- Der Resultatwert des Literals ist dann der von diesem Operator gelieferte Wert mit Typ `S` (d. h. Zeichenkettenliterale besitzen dann den Typ `S`).

Beispiel

```
str { (c:char) } -> (char* =
  h : char*?;
  t : char*?;
  {
    t =! ?t@tail =! char*(head = c);
    ?h \/ h =! ?t
  };
  ?h
)
```

- Der Operator `str` konstruiert eine Liste mit den Zeichen `c`, indem er jedes Zeichen am Ende der bereits konstruierten Liste anfügt, und liefert dann den Anfang der Liste zurück.

Exemplarische Verwendung

```
print "Hello!";          $$ H->e->l->l->o->!->

print [only] (s:char*) -> (bool =
  s : char*?; s =! s;
  while ?s do
    print only ?s.head;
    s =! (?s.tail)
  end;
  [ true | print 1:0 ]
);

print "Hello!"           $$ Hello!
```

- Da der implizit an Zeichenkettenliterale übergebene Operator `str` Resultattyp `char*` besitzt, besitzen auch Zeichenkettenliterale wie `"Hello!"` diesen Typ.
- Dementsprechend bezeichnet der Operator `print` in der ersten Zeile des Beispiels den am Ende von § 4.3.4 definierten Ausgabeoperator für Listen.
- Der anschließend definierte Operator `print` gibt nur die Zeichen der Liste `s` ohne Pfeile aus. Weil er spezieller als der generische Ausgabeoperator für Listen ist, wird er in der letzten Zeile des Beispiels bevorzugt.