5 Operator Values

5.1 Operators as Parameter Values

5.1.1 Basic Principle

- If the type of a parameter is an operator type (i. e., an operator declaration, cf. § 3.8.1) – no matter whether the parameter is anonymous and/or implicit or not –, an expression can be passed explicitly as an operand that returns a matching operator, i. e., an operator which can approximately replace the operator (cf. § 3.9.4) declared in the parameter type.

- In particular, it is possible to pass a constant that has previously been initialized with a matching operator.

  For that purpose, the rules for constant declarations mentioned in § 2.4 are extended as follows:
  If the type of the constant is an operator type, and the constant is not explicitly initialized, it is not initialized with a new synthetic value, but rather with the operator declared in the operator type. For example:

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

  Therefore, the value of the constant `square` is the operator $\bullet^2$ defined here.

- In contrast to § 3.11.2, only approximate replaceability is necessary when an operator is explicitly passed to a parameter, i. e., all names in the top level signature of the operator and the corresponding parameter are ignored according to § 3.9.4, such that an operator with different names, which might also appear at different positions in the signature, might be passed, too. For example (cf. § 3.8.2):

```
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
```

Even though the types of the constant `square` (i. e., `(int) "^2" -> (int)`) and the parameter `f` (i. e., `f (int) -> (int)`) have different names at different positions of their top level signature, the operator `square` can be passed to the parameter, because the parameter can be approximately replaced with the operator.

- In analogy to § 3.11.2, the passed operator might also be more general than the parameter, for example (cf. § 3.11.3):

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

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

The operator `generic square` can be passed to the parameter `f`, because every correct application of the parameter is also a correct application of the operator with the same type, provided the names of the parameter and the operator are ignored (and there is a multiplication operator for the respective type `T`).

- It is also possible – comparable to lambda expressions in other languages – to declare the passed operator directly at the point where it is needed.
  But because an operator declaration does not return the declared operator, but rather its type, a constant declaration is still necessary, which might also be anonymous, however. For example:

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

- To reduce the “syntactic overhead” even more, the names of the passed operator might be omitted (because they are meaningless anyway) as well as its parameter and result types (provided they can be deduced by compiler, which is possible in most cases). For example:

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

  The colon after the name of the parameter and the equals sign before the implementation of the operator are still necessary, however, because otherwise the parameter name could be interpreted as the type of an anonymous parameter, and the implementation expression could be interpreted as the result type of the operator.

5.1.2 Typical Examples from Functional Programming

Map

- `map xs using f` applies the function or rather operator `f` to all elements of the list `xs` (cf. § 4.3.4) and returns a new list containing the result values of the function invocations:

```
[(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
)
```

- For example:

```
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->
```

(Without the parentheses around `xs`, the last line could also be interpreted as the declaration of a constant with the names `print map xs using`.)

Filter

- `filter xs using f` applies the function `f` to all elements of the list `xs` and returns a new list containing all elements of `xs` for which the function invocation did not return nil:

```
[(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
)
```

  To make sure that the order of the invocations of `f` is the same as the order of the list elements, `f h` must be executed before the recursive invocation of `filter`.

- For example:

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

Fold

- `fold x and xs using f` combines the value `x` and the elements of the list `xs` left-associatively by successive invocations of the binary function `f` and returns the result of the last invocation. If the list `xs` is empty, the value `x` is directly returned:

```
[(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
)
```

- Accordingly, `fold xs and x using` combines the elements of the list `xs` and the value `x` right-associatively by successive invocations of `f` and returns the result of the last invocation. If the list `xs` is empty, the value `x` is directly returned again:

```
[(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
)
```

- For example:

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

- Difference between left and right fold:

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

`x + -y` is used instead of `x - y`, because the latter might also be interpreted as a recursive application of the operator passed to `fold` to the operands `x` and `-y`.

Remark

- Because lists, as commonly in functional programming, are defined recursively here – a list is either empty, or it consists of a first element `head` and a rest list `tail` –, the operators `map`, `filter`, and `fold` can be implemented recursively, too, which is also typical for functional programming:
  If the list is empty, the recursion ends, while the operator is recursively applied to the rest list otherwise. The “actual” operation is executed only for the first element of the current list in each recursive step.

- For very long lists, however, the recursion might cause a stack overflow because the MOSTflexiPL compiler – in contrast to compilers of many functional programming languages – does not automatically transform typical recursive patterns into equivalent iterative code.

5.2 Operators as Result Values

- If the result type of an operator declaration is an operator type, the implementation of the operator must return a matching operator, i. e., once again an operator that can approximately replace the operator declared in the result type.
- This might, for example, be defined as a local operator, again combined with an anonymous constant declaration. For example:

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

  For an `int` value `y`, `add y` returns an operator that maps an `int` value `x` to the `int` value `x + y`.

- Possible use: (cf. § 4.3.4, § 4.3.4 und § 5.1.2):

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

- To make `add` also usable for types other than `int` for which there is a matching addition operator (for example `rat`, cf. the task about rational numbers), it might be generalized as follows:

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

- Possible use then:

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

- To make `add` even more generally usable (e. g., if there is an addition operator that adds a `rat` and an `int` value and returns the result as a `rat` value), it can be defined even more generically:

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

- Possible use then:

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

- Definition of `mul` in analogy to `add`:

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

The result type of `mul` is the type of the locally defined operator (i. e., `(X) -> (Z)`), which can be automatically deduced by the compiler in the same way as the result type (`Z`) of this local operator.

- Composition of functions:

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

For a function `f` that maps a value of type `X` to a value of type `Y`, and a function `g` that maps a value of type `Y` to a value of type `Z`, `f before g` returns a function that first applies the function `f` to a value of type `X` and afterwards the function `g` to the resulting value.

- Possible use:

```
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->
```

- To allow combinations of multiple functions, •`before`• might be defined either left- or right-associative:

```
excl : (int) -> (int) before
            (: (int) -> (int) before : (int) -> (int)) end
```

- Possible use:

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

5.3 Operators as Variable Values

5.3.1 Basic Principle

- If the content type of a variable is an operator type, the variable might contain a matching operator, i. e., once again an operator that can approximately replace the operator defined in the content type. For example:

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

- The variable `logger` might contain any operator with parameter type `char*` (cf. § 4.3.4 and § 4.5) and result type `bool`.
  The parameter name `s` is necessary, because otherwise `(char*) -> (bool)` might also be interpreted as an application of the operator •`->`• defined in § 4.3.2, whose operands are both surrounded with redundant parentheses.
  Alternatively, the operator `(char*) -> (bool)` might be given an arbitrary name, e. g., `log (char*) -> (bool)`.

- For example:

```
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
```

- The operator `simple logger` prints the string `msg` normally, while the operator `colored logger` uses escape sequences to print it in red.
  (The character sequence `esc [ 3 1 m` switches the foreground color of an ANSI-compliant console to red, while `esc [ 3 9 m` switches back to the standard color. `esc` is the `escape` character with the decimal value 27.)
  Depending on some run time condition, either the operator `simple logger` or the operator `colored logger` is then stored in the variable `logger`.

- The operator contained in a variable might, for example, be passed as a parameter value to some other operator, for example:

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

  The operator defined in the first line, whose signature consists of two parameters only, applies the operator passed to the first parameter `f` to the string passed to the second parameter `s`.
  Therefore, the operator that is currently stored in the variable `logger` is applied to the string `"log output"` in the second line.

- The previously defined operator can also be defined generically:

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

5.3.2 Further Possibilities

- Because operators can be stored in variables, they can also be stored, e. g., in attributes of open types (cf. § 4.3.2) or similar data structures (cf. § 4.4).

- For example:

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

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

- `observer` defines a generic attribute (cf. § 4.3.4) that attaches an operator of type `(var:T?) -> (bool)` to every variable of type `T?`.
  (Again, the parameter name `var` is necessary to make sure that the second arrow in the expression `T? -> (var:T?) -> (bool)` unambiguously denotes an operator declaration and thus an operator type and that it cannot again be interpreted as an application of the arrow operator defined in § 4.3.2.
  However, the first arrow in this expression denotes just that arrow operator.)

- `observe var with f` then stores the operator `f` in the attribute `observer` of the variable `var`, for example:

```
x : int?;

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

- To achieve that an "observer" attached to a variable in that way is called for every subsequent assignment to the variable, the predefined assignment operator •=!• might be overridden (actually hidden) by a user-defined operator with the same signature (also see § 3.7.1):

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

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

To enable the user-defined operator to invoke the predefined operator that it hides itself, an "alias" for it with a different signature (•=!!• in that case) must be defined in advance.

- By that means, the operator that has been previously attached to the variable `x` with `observe x with ...`, will be called for every subsequent assignment `x =! ...`
- To make the same observer usable for multiple different variables, it receives the respective variable as a parameter value after it has been assigned a new value.
- If no observer has been attached to a variable `var`, `var.observer` returns nil, i. e., a nil operator that does not have an implementation and whose invocation, therefore, has no effect.