3.9 Comparison of Operators

3.9.1 Definitions

- ❐ An operator can be *replaced* (exactly) with another operator, if every unambiguous and correct application of the first operator is also an unambiguous and correct application of the second operator with the same type.

- ❐ Two operators are *congeneric*, if each of them can be replaced with the other.

- ❐ An operator is *more restricted* or *more specific* than another, if the first operator can be replaced with the second, but not vice versa.
  Then, the second operator is *more extensive* or *more general* than the first.

- ❐ Two operators are *incomparable*, if none of them can be replaced with the other.

- ❐ Two operators are *non-overlapping*, if no unambiguous and correct application of one of them is also an unambiguous and correct application of the other with the same type.

3.9.2 Rules

❐ If multiple congeneric operators are initially visible at some location, the one that has been defined or imported last *hides* all others, causing them to become invisible.

❐ If the same expression can be interpreted as an application of different operators that are visible (and not hidden) at the respective location and not excluded by an exclude specification of the operator above:

- ❍ If one of these operators is more specific than all others, the expression is interpreted as an application of this operator, i. e., the most specific operator is preferred.

- ❍ If none of these operators is more specific than all others, the expression is ambiguous.

  This ambiguity can be resolved by defining another operator which exactly covers the „intersection“ of all these operators, i. e., whose unambiguous and correct applications are also unambiguous and correct applications of all these operators with the same type.
  Because this additional operator is now more specific than all these operators, it is preferred according to the previous rule.

3.9.3 Examples

❐ In the following example, the first constant `x` is hidden by the second constant `x`, because both have the same type, causing the expression `print x` to be unambiguous.
The first constant `y`, however, is not hidden by the second constant `y`, because they have different types, causing the expression `print y` to be ambiguous:

```
x := 1; y := 1;
x := 2; y := ’y’;
print x;                    $$ 2
print y                     $$ Ambiguous
```

Many other uses of `y` are unambiguous, however, because only one of the two constants causes the respective expression to be type-correct, for example:

```
print y + 0;                $$ 1
print char int y            $$ y
```

- In the following example, the first operator `sum` is more specific than the other two, causing it to be preferred in expressions such as `sum of 1 and 2 end`.
  The second and third operator are congeneric, causing the third to hide the second, and therefore expressions such as `sum of 1 end` and `sum of 1 and 2 and 3 end` are unambiguous both before and after the definition of the third one:

```
sum of (x:int) and (y:int) end -> (int =          $$ Definition
  x + y                                           $$ of Operator 1
);

sum of (x:int) { and (y:int) } end -> (int =      $$ Definition
  s : int?; s =! x; { s =! ?s + y }; ?s           $$ of Operator 2
);

print sum of 1 and 2 end;              $$ Use of Operator 1
print sum of 1 and 2 and 3 end;        $$ Use of Operator 2

sum of { (x:int) and } (y:int) end -> (int =      $$ Definition
  s : int?; s =! y; { s =! ?s + x }; ?s           $$ of Operator 3
);

print sum of 1 and 2 end;              $$ Use of Operator 1
print sum of 1 and 2 and 3 end         $$ Use of Operator 3
```

❐ In the following example, the second operator is more specific than the first, causing it to be preferred after its definition for swapping two `int` variables:

```
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =          $$ Definition
  z := ?x; x =! ?y; y =! z; y                    $$ of Operator 1
);

a : bool?; a =! true;
b : bool?; b =! false;
a <-> b;                                    $$ Use of Operator 1
u : int?; u =! 1;
v : int?; v =! 2;
u <-> v;                                    $$ Use of Operator 1

(x:int?) "<->" (y:int?) -> (int? =               $$ Definition
  x =! ?x + ?y; y =! ?x - ?y; x =! ?x - ?y; y   $$ of Operator 2
);

a <-> b;                                    $$ Use of Operator 1
u <-> v                                     $$ Use of Operator 2
```

3.9.4 Approximate Replaceability

Principle

❒ In certain situations, it is not necessary that an operator can be exactly replaced with another one, because the names of the operators have no or little relevance there and therefore can be ignored at least to some extent.

Preliminary Definition

❒ An operator can be *approximately* replaced with another one, if it can be exactly replaced with the other when the names in the top level signature of both operators are ignored.

Remarks

❒ The definition is preliminary and might be adjusted in the future, because it is inappropriate for some cases yet.

❒ Furthermore, the implementation of the concept in the current version of the compiler is still erroneous, i. e., it its behaviour is not correct in some cases yet.

❒ For the examples shown in the following, however, it works as desired.

Examples

❒ In the following example, each of the three operators can be approximately replaced with the other two (various operators used here are defined in task sheets):

```
(x:int) "²" -> (int = x * x);
(n:int) "!" -> (int = if n <= 1 then 1 else (n-1)! * n end);
"|" (x:int) "|" -> (int = if x >= 0 then x else -x end)
```

❒ The second and third operator `sum` from § 3.9.2 can be approximately replaced with the following operator `avg` and vice versa:

```
avg of (x:int) { and (y:int) } end -> (int =
  s : int?;
  s =! x;
  { s =! ?s + y };
  n := 1 + {1};
  ?s : n
)
```

3.10 Forwarding of Operator Applications

3.10.1 Problem

❒ To avoid code replication, an operator such as `avg` in § 3.9.4 should be able to invoke the second or third operator `sum` from § 3.9.2 to compute the sum of all its operand values.

❒ With the bracket operators mentioned in § 3.3, however, it is only possible to access the values of a repeatable parameter one after the other, but not to forward all of them to another operator.

3.10.2 Solution

❒ In the implementation of each operator, there is another bracket operator `<>(•)` whose operand might be an application of any other operator with which the current operator could be approximately replaced if its result type would be the type of this operand.

❒ At run time, the whole current operator application is in essence forwarded to this other operator, and the value returned by it is returned by the application of the bracket operator.
The operand of the bracket operator is only needed to identify the other operator and is never evaluated at run time.

3.10.3 Example

❒ Forwarding of the applications of `avg` to the second or third operator `sum`:

```
avg of (x:int) { and (y:int) } end -> (int =
  <>(sum of 1 end) : (1 + {1})
)
```

3.10.4 Partial Forwarding

❒ In some cases it is desirable to forward only a part of an operator application to another operator.

❒ Therefore, in addition to the brackets in the signature of an operator that are mentioned in § 3.2, round brackets with just one alternative are allowed if they have a denomination `<names>`.

❒ For each bracket of this kind, there is a corresponding bracket operator `<names>(•)` in the implementation of the operator, which works in analogy to the bracket operator mentioned in § 3.10.2, but forwards only the part of the current operator application that belongs to the subsignature in this bracket.

3.10.5 Example

```
$$ Compute expression in "additive normal form".
calc <ab>( [minus] (a:int) { times [minus] (b:int) } )
    { plus <cd> ( [minus] (c:int) { times [minus] (d:int) } ) } end
-> (int =
  $$ Compute product.
  prod <mx>[minus] (x:int) { times <my>[minus] (y:int) } end
  -> (int =
    p : int?; p =! <mx>[-x | x]; { p =! ?p * <my>[-y | y] }; ?p
  );
  $$ Compute first product with factors a and possibly b.
  r : int?;
  r =! <ab>(prod 1 end);
  $$ ^
  $$ ^
  { r =! ?r + <cd>(prod 1 end) };
  ?r
);

$$ Exemplary uses.
print calc 2 times minus 3 plus 4 times 5 end;  $$ 14
print calc 2 times 3 plus minus 4 times 5 end   $$ -14
```