5.5 Value assignment and comparison

In section , we have described assignment and comparison for references. In summary, we may assign a reference `r2` to `r1` (`r1 := r2`): `r1` then refers to the same object as `r2`. We do not copy the content of the object referred by `r2` to the object referred by `r1`. Two different reference data items may refer to the same object.

Similarly for a comparison, `r1 = r2` is true if `r1` and `r2` refer to the same object – and `r1 <> r2` is true if they refer to different objects.

For primitive value types the situation is different. Value objects are just needed for representing values – it is the content of a value object that is important and not the value object itself.

If `v1` and `v2` are different data-items of type `float`, we may assign `v2` to `v1` (`v1 := v2`) and the value hold by `v2` is now also hold by `v1`. It is *not* the case that `v1` and `v2` refer to the same value object.
The same applies to comparison, the comparison `v1 = v2` is true if `v1` and `v2` represent the same value. This is illustrated by the following example:

```
v1,v2: var float
b: var Boolean
v1 := 3.14
v2 := 7.5
b := v1 = v2  -- now b = false since v1 and v2 represent different values
b := v1 <> v2 -- now b = true
v2 := v1      -- now v1 and v2 represent the same value, 3.14
b := v1 = v2  -- now b = true
```

If `v1` and `v2` are two different value objects of the same type, assignment `v1 := v2` implies that the local data-items in `v2` is copied to the corresponding data-items in `v1`.

For a comparison `v1 = v2`, is true if the local data-items of `v1` are equal to the corresponding local data-items of `v2`. If the local data-item is a compound value, this applies recursively.

We want assignment and companions for value objects in general to work as for primitive value types. Two value objects of type `Point` may represent the same point or different points, and this must be reflected in assignment and comparison. Consider the following example

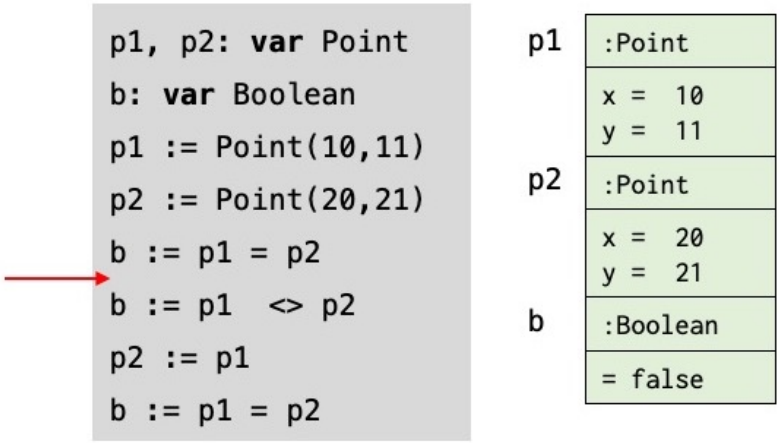
```
class Point(x,y: var Integer): Value
   ...

p1, p2: var Point
b: var Boolean
p1 := Point(10,11)
p2 := Point(20,21)
b := p1 = p2      -- now b = false since p1 and p2 represent different points
b := p1 <> p2     -- now b = true
p2 := p1          -- now p1 and p2 represent the same point(10,11)
b := p1 = p2      -- now b = true
```

The next list of snapshots illustrates the effect of assignment and comparisons on value objects:

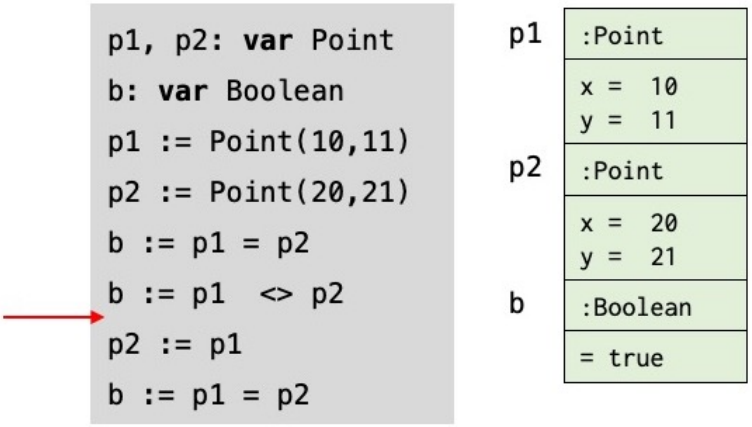
The first snapshot shows the situation after `b := p1 = p2` – marked by a red arrow.

- The assignment `p1 := Point(10,11)` has been executed;
- the value of `p1` is `p1.x = 10` and `p1.y = 11`;
- `p2 := Point(20,21)` has been executed;
- the value of `p2` is thus `p2.x = 20` and `p2.y = 21`;
- `b := p1 = p2` has been executed;
  - this includes evaluation of the comparison `p1 = p2` evaluation to `false`;
  - the value of `b` is thus `false`.



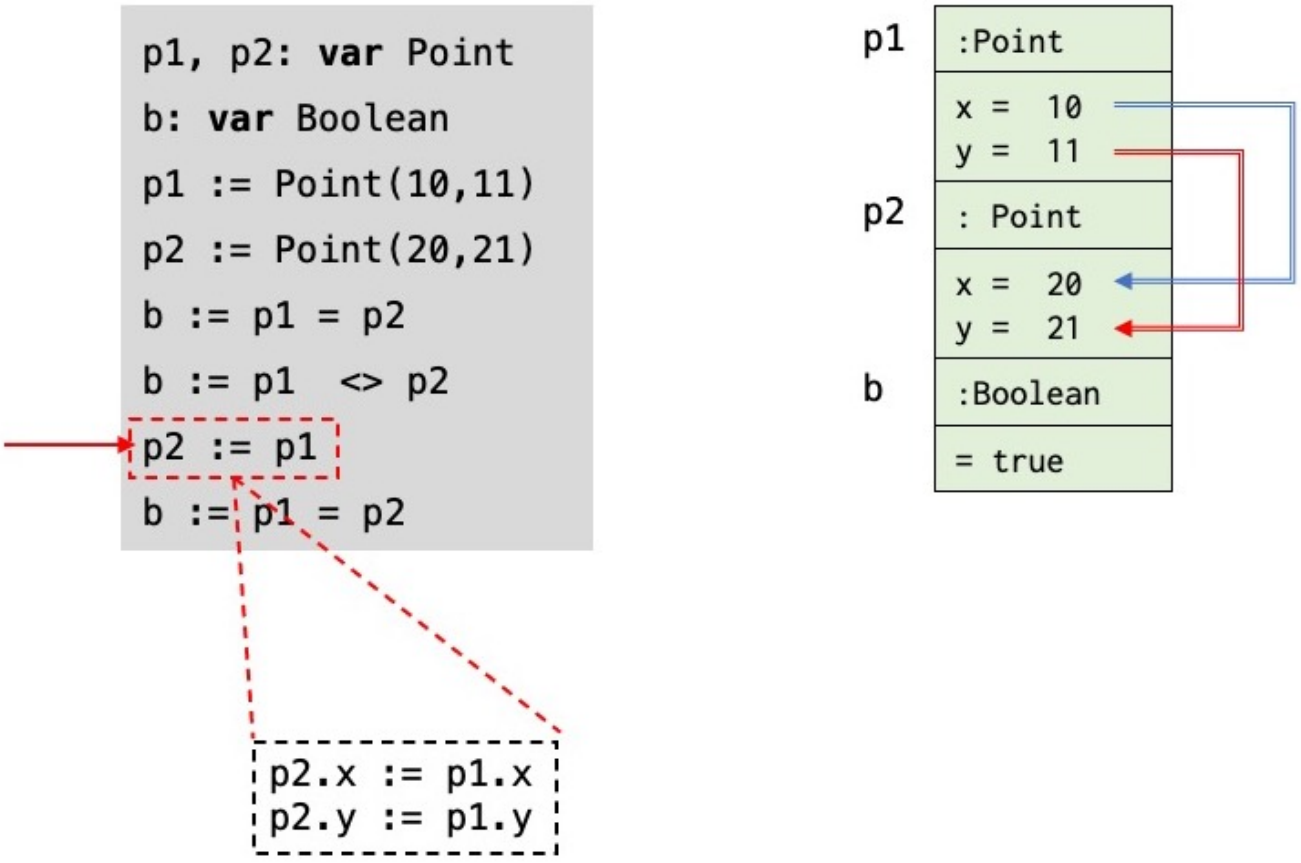
This snapshot shows the situation after the assignment `b := b1 <> b2`.

- This includes evaluation of the comparison `b1 <> b2` evaluating to `true`;
- the value of `b` is thus now `true`



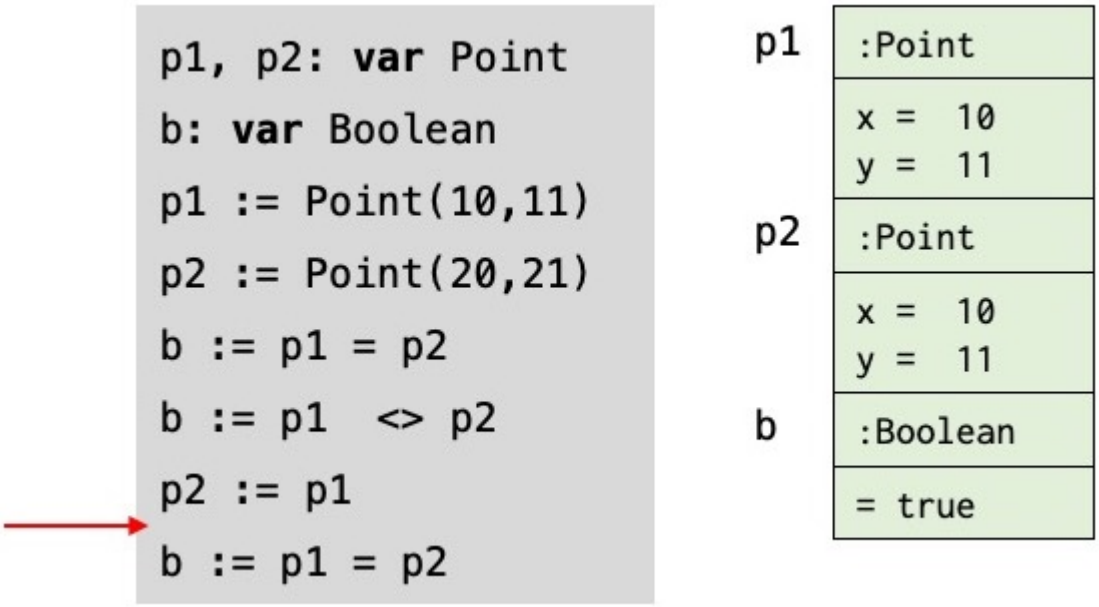
This snapshot shows the immediately before execution of the assignment `p2 := p1`.

- Execution of `p2 := p1` corresponds to executing the assignments:
  - `p2.x := p1.x`
  - `p2.y := p1.y`



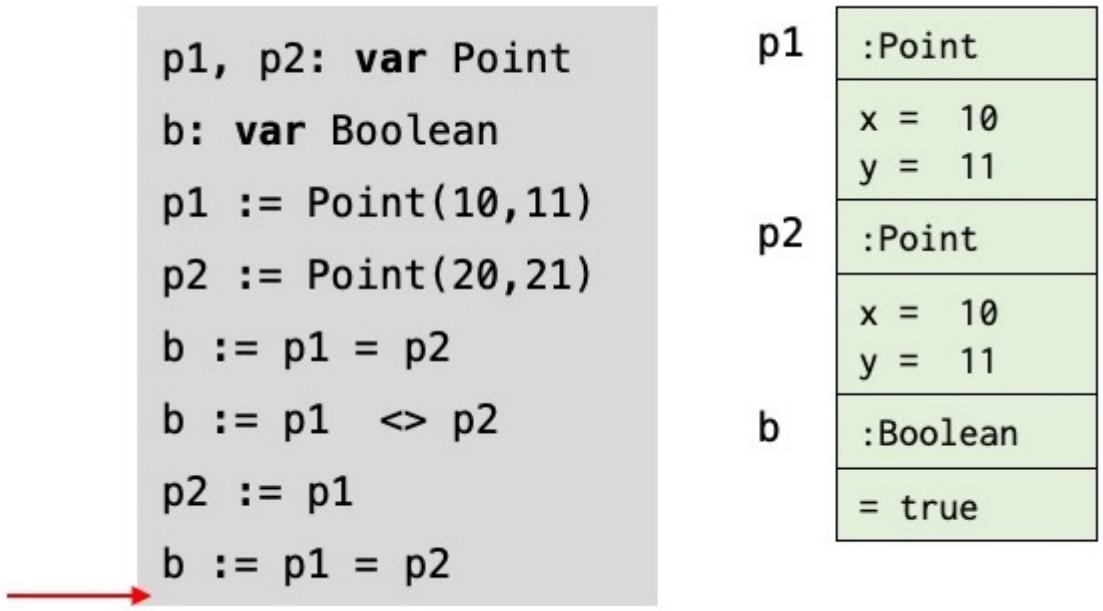
This snapshot then shows the situation after execution of the assignment `p2 := p1`.

- The value of `p2` is now `p2.x = 10` and `p2.y = 11`;



The final snapshot shows the situation after the assignment `b := p1 = p2`.

- The right side of the assignment is a comparison `p1 = p2` that values to `true`;
- the value of `b` is now `true`.

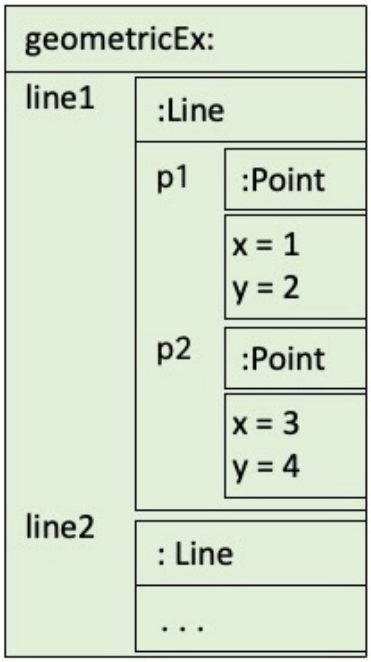


Here we have a more elaborated example. We define class `Line` as a value object with data-items of type `Point`:

```
class Line(p1,p2: var Point): Value
   ...
```

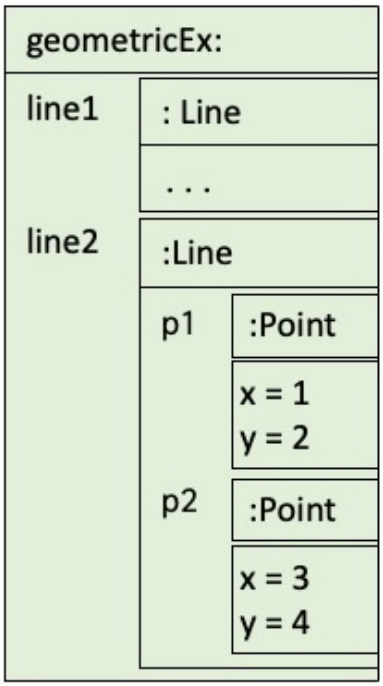
The following snapshots of object `geometricEx` illustrates the effect assignment of `Line` objects. The first snapshot shows the situation after `line1 := Line(Point(1,2), Point(3,4))` has been executed. As can be seen, `line1.p1.x = 1`, `line1.p1.y = 2`, `line1.p2.x = 3`, and `line1.p2.y = 4`.

```
geometricEx: obj
   line1,line2: var Line
   line1 := Line(Point(1,2), Point(3,4))
   line2 := line1
```



The next snapshot shows the situation after the assignment `line2 := line1`. The assignment implies that the local data-items of `line1` is copied to the corresponding data-items of `line2`. Since the local data-items are composite values of type `Point`, the copying is recursive in the sense that the local data items of the `Point` objects are copied to the corresponding `Point` objects in `line2`. As can be seen, `line2.p1.x`, etc. now all have the values as the corresponding variables in `line1`:

```
geometricEx: obj
   line1,line2: var Line
   line1 := Line(Point(1,2), Point(3,4))
   line2 := line1
```



Value parameter transfer

Invocation of a method like `anAccount.deposit(500)` involves the transfer of the argument 500 to the parameter `amount` of the `deposit` object generated as part of the invocation – this is called *parameter transfer*. Parameter transfer is similar to assignment in the sense that the argument 500 is assigned to the parameter `amount`.

In a similar way generation of an object like `Point(2,3)` implies that the arguments 2 and 3 are *assigned* to the parameters `x` and `y` of the generated `Point` object as described above.

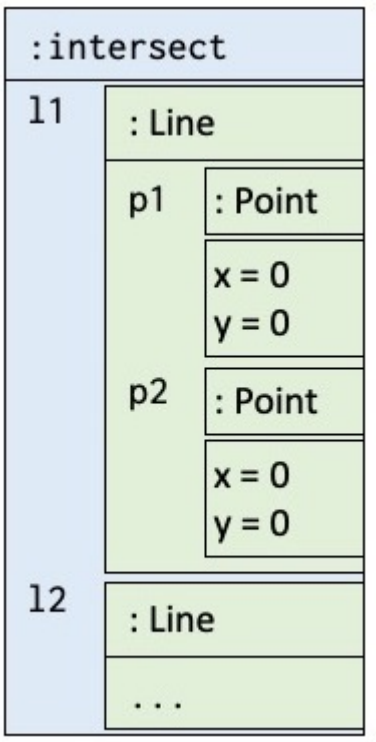
The generation of a `Line` object as in `Line(Point(1,2), Point(3,4))` involves generation of two `Point` objects where the arguments 1, 2 and 3, 4 are assigned to the `x` and `y` parameters of the two new `Point` objects and these `Point` objects are then subsequently assigned to the `p1` and `p2` parameters of the new `Line` object.

Next we illustrate parameter transfer for a `Line` object. We use a method that computes the intersection of two lines as an example although we leave the computation as an exercise. For simplicity, we assume there is an intersection point between the two lines.

```
intersect(l1,l2: var Line) -> p: var Point:
   p := ...
```

Below we have extended `geometricEx` and the first snapshot shows the situation *during* the execution of `intersectingPoint := intersect(line1, line2)` and at the start of the invocation of `intersect` (marked by red) at the point where a method object with default values has been generated:

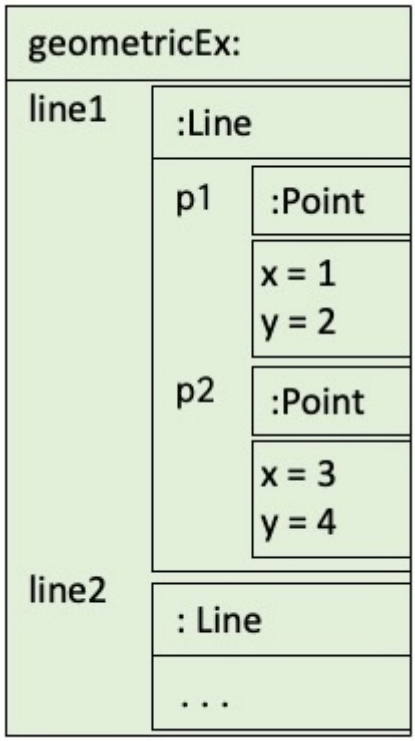
```
geometrixEx: obj
   line1, line2: var Line
   line1 := Line(Point(1,2), Point(3,4))
   line2 := Line(Point(5,6), Point(7,8))
   intersectingPoint: var Point
   ...
   intersectingPoint := intersect(line1, line2)
```

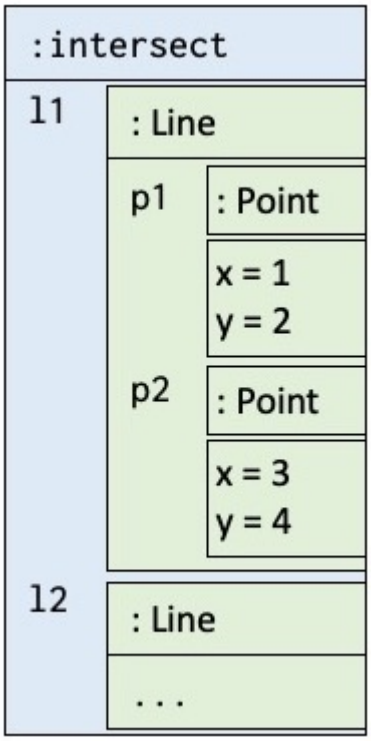


After the intersect method object has been generated the arguments `line1` and `line2` are passed to the method object. This involves two assignments `intersect.l1 := line1` and `intersect.l2 := line2`.

The assignment `intersect.l1 := line1` implies that the local data-items of `line1` is copied to the corresponding data-items of `l1` in the `intersect` method object. Since the local data-items are composite values of type `Point`, the copying is recursive in the sense that the local data items of the `Point` objects are copied to the corresponding `Point` objects in `l1`. Similarly for the second argument `line2`.

The next snapshot shows the `geometricEx` object and the `intersect` method object after the arguments have been transferred – for simplicity only `line1` and `l1` are shown:





Aliasing

As a consequence of the above rules for assignment it is not possible to have two or more names that denote the same value object as is the case for objects.

The fact that you can denote the same object with two or more names is called *aliasing*. Aliasing may complicate programming since the state of an object may be modified from different places in a

program – the programmer therefore has to be careful when using references. As we argue below, we need aliasing for objects.

However, we do not want aliasing when dealing with values since this does not make sense and may easily lead to unexpected results. It is thus not possible to have two or more names denoting the same value object.

In the following example, we illustrate the difference between data items representing values and data items representing references with respect to aliasing. First we consider value objects:

- After the assignment `p2 := p1`, `p2 = Point(1,2)`.
- After the assignment `p1 := Point(3,4)`, `p1 = Point(3,4)`, and `p2` is unaffected and we still have `p2 = Point(1,2)`.

```
p1,p2: var Point
p1 := Point(1,2)
p2 := p1
p1 := Point(3,4)
-- p2 = Point(1,2)
```

The next example illustrates arising for references:

- After the assignment `acc2 := acc1`, `acc1` and `acc2` refer to the same `Account` object.
- After the assignment `acc1.balance := 250`, we also have `acc2.balance = 250`, since `acc2` refers to the same object as `acc1`.

```
acc1, acc2: ref Account
acc1.balance := 150
acc2 := acc1
acc1.balance := 250
-- acc2.balance = 250
```

Rationale

As mentioned in section , we distinguish phenomena representing physical entities local accounts, vehicles, customers, etc. and properties of physical entities like balance of an account, length of a

vehicle and the name of a customer.

Objects represent physical entities and we may have several references to a given object. A given vehicle may be referred to from a vehicle registration system, an insurance company, the owner of the vehicle, etc. We thus want to be able to represent having several references to the same object in our computerized models. Assignment and comparison of references are thus about references and not the state (data-items) of the objects.

The properties of physical entities are often represented by values of some type and as said above, from a modeling point of view, it is the values that are relevant and not the value objects holding the values.

This is in contrast to objects representing physical entities – here the objects are relevant with respect to modeling. The state of a given object with respect to the datums hold by its data items are of course also important.