5.1 Reference data-items

Description

In this section, we further describe data-items holding references to objects.

We have in previous sections like seen declarations of the form:

```
JohnSmithProfile: obj Customer("John Smith")
```

This declaration implies the generation of an instance of the `Customer` class with `"John Smith"` as the actual parameter. The data-item `JohnSmithProfile` is a constant reference that refers to this object during the life-time of the program execution.

We have also seen examples of declarations of the form:

```
aCustomer: ref Customer
```

The data-item `aCustomer` is a reference variable that may refer to different `Customer` objects during the life-time of the program execution. Initially it holds the reference `none`, which means that it refers to no object.

We have also seen examples of assignment of references that has the effect that two or more references may refer to the same object.

In the next sections, we describe reference assignment in details. We also describe comparisons of references, parameter transfer of references and the type rules for reference assignment and comparison.

Reference assignment

Here we will summarise assignment of references in general. We use class `Customer` for the example and we use the following ghost object because in a program for a real bank there would be no object with the description below:

```
aGhost: obj
   JohnSmithProfile: obj Customer("John Smith")
   LizaJonesProfile: obj Customer("Liza Jones")
   customerA, customerB: ref Customer
   customerA := JohnSmithProfile
   customerB := LizaJonesProfile
   customerA := customerB
```

Figure 5.1.1 Customer references

In the above example, we have two `Customer` objects `JohnSmithProfile` and `LizaJonesProfile`, and two reference variables `customerA` and `customerB`. The following snaphots illustrates the effect of reference assignments.

The first snapshot shows the situation after generation of `aClerk` – marked by the red arrow (–>). Here `JohnSmithsProfile` refers to `Customer("John Smith")` and `LizaJonesProfile` refers to `Customer("Liza Jones")`. The reference variable `customerA` and `customerB` are both none:

```
 aGhost: obj
    JohnSmithProfile: obj Customer("John Smith")
    LizaJonesProfile: obj Customer("Liza Jones")
    customerA, customerB: ref Customer
--> customerA := JohnSmithProfile
```

```
    customerB := LizaJonesProfile
    customerA := customerB
```

Image not found or type unknown

Snapshot A

The next snapshot shows the situation after the assignment `customerB := LizaJonesProfile`. As can be seen, `customerB` and `LizaJonesProfile` now both refer to the same object:

```
 aGhost: obj
    JohnSmithProfile: obj Customer("John Smith")
    LizaJonesProfile: obj Customer("Liza Jones")
    customerA, customerB: ref Customer
    customerA := JohnSmithProfile
    customerB := LizaJonesProfile
--> customerA := customerB
```

Image not found or type unknown

Snapshot B

The final snapshot shows the situation after execution of `customerA := customerB`. As can be seen, `customerA` and `customerB` now both refer to `LizaJonesProfile`.

```
 aGhost: obj
    JohnSmithProfile: obj Customer("John Smith")
    LizaJonesProfile: obj Customer("Liza Jones")
    customerA, customerB: ref Customer
    customerA := JohnSmithProfile
    customerB := LizaJonesProfile
    customerA := customerB
-->
```

Image not found or type unknown

Snapshot C

Reference comparison

We may also compare references using = (equality) and <> (inequality):

If E1 and E2 are expressions that evaluates to references R1 and R2, then the expression E1 = E2 is true if and only if R1 and R2 refer to the same object. If R1 and R2 refer to different objects then E1 = E2 evaluates to false.

Similarly the expression E1 <> E2 is true if and only if R1 and R2 does not refer to the same object. If R1 and R2 refer to the same object then E1 = E2 evaluates to false.

Below, we show the value of some reference expressions using = (equality) and <> (inequality) at Snapshot B and Snapshot C above.

For the situation at Snapshot B above, we have the following:

```
JohnSmithProfile = LizaSmithProfile    -- false, they refer to different objects
JohnSmithProfile <> LizaSmithProfile   -- true, they refer to the same object
JohnSmithProfile = customerA           -- true, they refer to the same object
JohnSmithProfile <> customerB          -- false, they refer to different objects
```

The first comment `-- false, they refer to different objects` is meant to say that the expression `JohnSmithProfile = LizaSmithProfile` evaluates to the value `false`.

The situation at Snapshot C after the assignment `customerA := customerB` is as follows:

```
JohnSmithProfile = customerA   -- false
customerA = customerB          -- true
LizaJonesProfile = customerA   -- true
```

Assignment between data items being references is called r*eference assignment* and comparison of references is called *reference comparison*.

Reference assignment and reference comparison is fundamentally different from assignment between data items representing values.

The `withdraw` method has a statement:

```
newB := balance
```

Here the value hold by `balance` is copied to `newB`, which then holds the same value as `balance`. The data items `newB` and `balance` are not references to some objects. As we shall se in section X, they are a special kind of objects called *value objects* that may represent values – in section , we describe *value assignment* and *value comparison*.

Reference parameter passing

Passing a parameter as part of a method invocation or class invocation is similar to assignment in the sense that the actual parameter is assigned to the formal parameter of the method or class respectively.

Consider the following example:

```
account_1010: obj Account(JohnSmithsProfile)
```

Here an instance of `Account` is generated with `JohnSmithsProfile` as the actual parameter. First an instance of `Account` is generated and then `JohnSmithsProfile` is assigned to the `owner` reference variable of this `Account` object. This may be illustrated by the following code sketch:

```
anAccount: ref Account
anAccount := Account   -- generate the/an Account object
anAccount.owner := JohnSmithsProfile
account_1010 := anAccount
```

Note that, the statement `anAccount := Account` is only for illustrative purposes – it is not possible to write this statement in qBeta, since an actual parameter must be supplied when `Account` has a parameter (here `owner`)
. +++ Vi skal arbejde med formuleringen her – generate omfatter normal også parameter overførsel så måske et andet ord her?

Type rule for reference assignment and comparison

As shown above, we may assign a reference to a `Customer` object to a reference variable that has the type `Customer`. It is not possible to assign a reference to an `Account` object to the reference variable `aCustomer`.

In general the type of an expression in an assignment must be the same as the type of the reference variable being assigned to. This is also the case for passing an expression as an argument to a parameter of a method being a reference.

The above rule does also apply to comparisons using = (equality) and <> (inequality) where both arguments must be of the same type

Consider the following example:

```
aCustomerA, aCustomerB: ref Customer
anAccountA, anAccountB: ref Account
B: var Boolean
aCustomerA := Customer("John Smith")   -- legal
anAccountA := Account(aCustomerA)      -- legal
aCustomerA := anAccountA               -- illegal
anAccountB := Account(anAccountA)      -- illegal
B := aCustomerA = aCustomerB           -- legal
B := aCustomerA <> anAccountA          -- illegal
```

The assignment `anAccountB := Account(anAccountA)` is illegal since the `owner` parameter of `Account` is of type `Customer` whereas the argument `anAccountA` is of type `Account`.

The purpose of the type rule is two fold: from a programming and modeling point of view it does not make sense to allow assignments like `aCustomerA := anAccountA`.

Secondly the type rule is necessary to prevent errors at run-time. Assume that we allow the assignment then we may write code as

```
aCustomerA := anAccountA
aCustomerA.addAccount(JohnSmithProfile)
```

This does not makes sense since an `Account` object does not have an `addAccount` method.

In chapter , we extend the type rule for assignment, parameter transfer and comparison.