8.1.2 Extended type rule

Description

Subclasses makes it possible to organize classes in a hierarchy where common attributes are defined in a superclass and more specialised attributes in subclasses. In addition, it is possible to write code that is general for objects being instances of different classes and/or subclasses in a hierarchy. For the bank example we may write code that works for all kinds of accounts. In order to be able to do this, we extend the type rule described in section .

Consider the example:

```
anAccount: ref Account
aSavingsaccount: ref SavingsAccount
aCreditAccount : ref CreditAccount
aSavingsAccount := SavingsAccount(JohnSmithProfile)
aCreditAccount := CreditAccount(JohnSmithProfile)
```

In the example above we create objects of type `SavingsAccount` and `CreditAccount` and assign the references to these objects to the reference variables `aSavingsAccount` and `aCreditAccount` respectively. This all in accordance with the type rule described in section .

We may extend the type rule as shown in the next example:

```
anAccount := aSavingsAccount
anAccount.deposit(400)
anAccount := aCreditAccount
andAccount.withdraw(299)
```

We assign the reference held by a `aSavingsAccount` to the reference variable `anAccount`. We then invoke the `deposit` method using `anAccount`. This is possible since the `deposit` method is described in class `Account` and all attributes of `Account` are also attributes of class `SavingsAccount`.

We may in a similar way assign the reference held by `aCreditAccount` to `anAccount` and subsequently invoke an `Account` method – here `withdraw`.

In general the extended type rule is defined as follows

> A reference expression of type `TT` may be assigned to a reference variable of type `T` if `T` is a (direct or indirect) superclass of `TT`.

If a reference variable `r` is of type `T` all attributes defined in class `T` may be accessed, but even if `r` refers to an object of type `TT`, attributes defined in class `TT` cannot be accessed.

For the account objects this means that using `anAccount` we may access the attributes defined in class `Account`. We may not access the ones defined in class `SavingsAccount` or `CreditAccount`, since we in general do not know if `anAccount` refers to an instance of `Account`, `SavingsAccount` or `CreditAccount`.

We may also assign a reference to a singular object derived from a class (e.g. `Account`) to a reference variable typed by this class (here `Account`) as shown in the following ghost example:

```
anAccount: ref Account
aSpecialAccount: obj Account
   withdraw::<
       -- create alert
...
anAccount := aSpecialAccount
```

As can be seen, `aSpecialAccount` may be assigned to `anAccount` which is of type `Account`.

The rationale for the extended type rule is the same as for the simple type rule described in section : it makes sense from a programming and modeling point of view and the compiler may check the rule and thus prevent errors of this kind a run-time.

It is however, possible to write a so-called *reverse assignment* like

```
aSavingAccount := anAccount
```

Here the compiler will check during execution of the program (at run-time) that `anAccount` does refer to an object of type `SavingsAccount`. The programmer may know that this is the case, but the compiler in general does not know.

One may discuss whether or not this is good from a programming and modeling point of view – with respect to safety, a run-time error may happen here.