Assignment 5: SCOOP type system

ETH Zurich

1 Subtyping

1.1 Background

Have a look at the attributes shown in listing 1.

Listing 1: Attributes

```
px: PROCESSOR
py: PROCESSOR

a: separate X
b: separate <px> X
c: separate <py> X
d: X
e: detachable separate X
f: detachable separate <px> X
g: detachable X
```

1.2 Task

Decide whether the following attachments are valid or not. Justify your answer.

1. $a := b$
2. $a := d$
3. $b := a$
4. $b := c$
5. $b := d$
6. $d := a$
7. $d := b$
8. $a := e$
9. $e := a$

1.3 Solution

1. The assignment $a := b$ is valid. All type components of b are conformant to the type components of a.

2. The assignment $a := d$ is valid. All type components of d are conformant to the type components of a.

3. The assignment $b := a$ is invalid. The $\top$ processor tag does not conform to the explicit processor tag.

4. The assignment $b := c$ is invalid. The two explicit processor tags are not conformant to each other. The two explicit processor tags denote different processors.

5. The assignment $b := d$ is invalid. The non-separate processor tag does not conform to the explicit processor tag. The explicit processor tag denotes a processor different than the current processor.

6. The assignment $d := a$ is invalid. The $\top$ processor tag does not conform to the non-separate processor tag. Statically the $\top$ processor tag can denote any processor.

7. The assignment $d := b$ is invalid. The explicit processor tag does not conform to the non-separate processor tag. The explicit processor tag denotes a processor different than the current processor.

8. The assignment $a := e$ is invalid. A detachable type does not conform to an attached type.

9. The assignment $e := a$ is valid. All type components of a are conformant to the type components of e.

2 Valid targets

2.1 Background

Have a look at listing 2.

Listing 2: Enclosing Feature

```
p: PROCESSOR

r (a: detachable separate X; b: separate <p> X; c: separate X)
  local
    d: separate <p> X
    e: separate <c.handler> X
    f: separate X
  do
    ...
  end
```

Imagine that the class *X* has a function *g*: *X* and a procedure *do_something*.

2.2 Task

Decide for each of the following feature calls, whether the calls are valid or not when they appear in feature r of listing 2.

1. $c.do\_something$
2. $c.g.do\_something$
3. $e := c;\ e.do\_something$
4. $f := c;\ f.do\_something$
5. $a.do\_something$
6. $d := b;\ d.do\_something$

2.3 Solution

1. The call $c.do\_something$ is valid. The target c is attached and it appears as a formal argument in the enclosing routine.
2. The call $c.g.do\_something$ is valid. The expression c has an implicit type $(!, c.handler, X)$. The result type combiner yields $(!, c.handler, X)$ as the type of $c.g$. Thus the target $c.g$ is attached and has a qualified explicit processor tag denoting an attached formal argument of the enclosing routine.
3. The call $e.do\_something$ is valid. The target e is attached and has a qualified explicit processor tag denoting an attached formal argument of the enclosing routine.
4. The call in $f := c;\ f.do\_something$ is invalid. The entity f is separate and does not correspond to any of the attached formal arguments in the enclosing routine. At runtime the entity f will be attached to a controlled object. Therefore an object test would help to make the call valid.
5. The call $a.do\_something$ is invalid. The target a is not attached.
6. The call $d.do\_something$ is valid. The target d is attached and it has the same same unqualified explicit processor tag as one of the attached formal arguments in the enclosing routine.

3 Separate generics or generic separate?

3.1 Background

The interplay between generics and separate types are important to understand, and enforce a good understanding of the type system.

3.2 Task

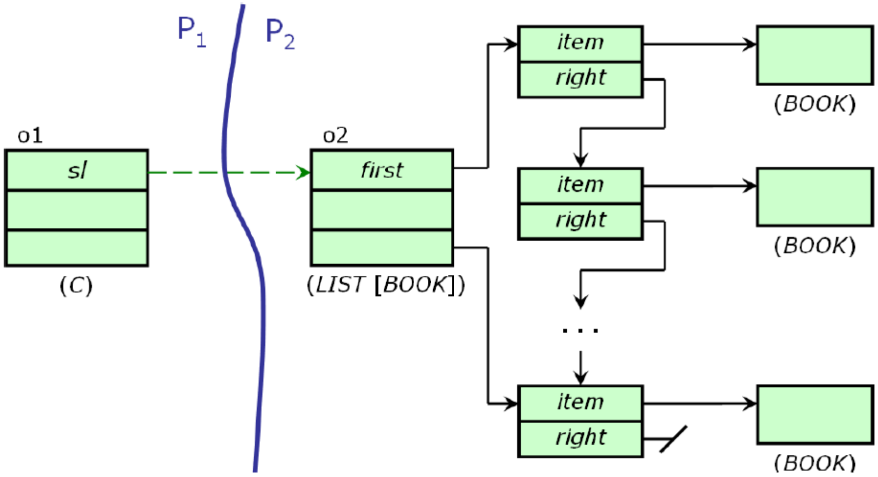
Consider the differences between:

- **separate** *LIST* [*BOOK*]
- *LIST* [**separate** *BOOK*]

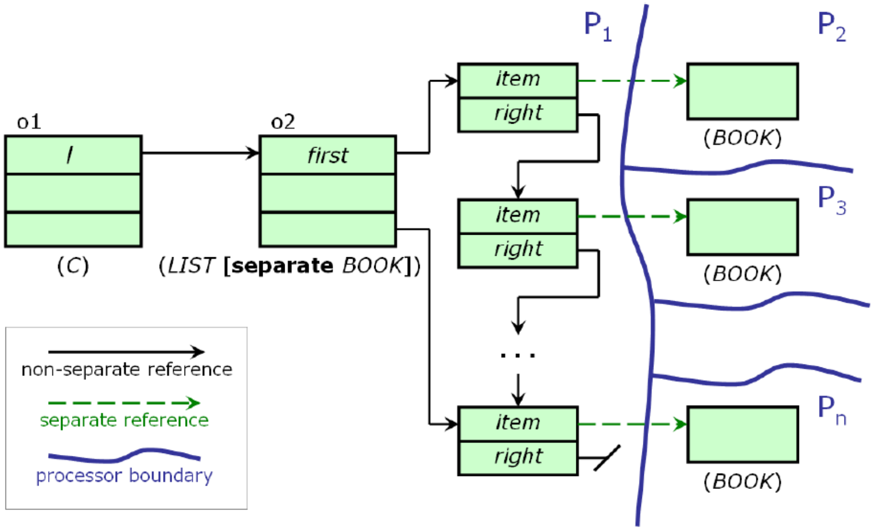
Explain the distinction using the object/processor diagram.

3.3 Solution

A separate list of books:



A list of separate books:



4 Basic library: type combiner

4.1 Background

Consider the classes in listing 3. These classes belong to a basic library implementation.

Listing 3: Basic Library

```
class LIST[G]
  feature
    last: G
          -- Last element.

    put(a_element: G)
```

```
        -- Add the element to the list.
    do
        ...
    end
end

class LIBRARY
  feature
    books: LIST[separate BOOK] -- Books.
end
```

4.2 Task

What is the result type of *books*. *last* from the perspective of the library? What is the type of an actual argument in the call *books*. *put* (...) from the perspective of the library? Justify your answer.

4.3 Solution

The type of the target *books* is $(!, \bullet, LIST[(!, \top, BOOK)])$. The result type of *last* is $(!, \top, BOOK)$. As a result one gets $(!, \bullet, LIST[(!, \top, BOOK)]) * (!, \top, BOOK) = (!, \top, BOOK)$. The type of the formal argument of *put* is $(!, \top, BOOK)$. Thus the combination yields $(!, \bullet, LIST[(!, \top, BOOK)]) \otimes (!, \top, BOOK) = (!, \top, BOOK)$.

5 Stack library: type combiner

5.1 Background

Consider the alternative stack based library implementation shown in listing 4.

Listing 4: Stack Library

```
class LIST[G]
  feature
    last: G -- Last element.
end

class STACK[G]
  feature
    top: G -- Top element.
end

class LIBRARY
  feature
    books: LIST[STACK[separate BOOK]] -- Books.
end
```

5.2 Task

What is the result type of *books*. *last* . *top* from the perspective of the library? Justify your answer.

5.3 Solution

The result type can be determined by applying the result type combiner several times as shown in the following.

$$(!, \bullet, LIST[B]) * \overbrace{(!, \bullet, STACK[A])}^{B} * \overbrace{(!, \top, BOOK)}^{A} =$$

$$(!, \bullet, STACK[A]) * \overbrace{(!, \top, BOOK)}^{A} = (!, \top, BOOK)$$