Composite and product types

Contents:

- Composite types
  - Constructing a composite type
    - Constructor
  - Operators
  - Comparison
- Product types
- Examples of specification

Composite types

A composite object usually contains several fields, each describing the different aspect of the object. A composite object is like a record in Pascal and structure in C. A composite type provides a set of composite objects.

A composite type is constructed using the type constructor: composed of ... end.

The general form of a composite type declaration:

```
A = composed of
      f_1: T_1
      f_2: T_2
        ...
      f_n: T_n
    end
```

where f_i (i =1,...,n) are variables called fields and T_i are their types. Each field represents an attribute of the composite object of the type. A value of a composite type is called composite object (or composite value).

A variable co of composite type A can be declared in one of the following forms:

(1) co: A

(2) co: composed of
f_1: T_1
f_2: T_2
...
f_n: T_n
end

Example: type declaration:

```
Account = composed of
            account_no: nat
            password: nat
            balance: real
          end
```

Variable declaration:

```
account: Account
```

(1) Constructor

Only one constructor known as make-function is available for composite types. The format:

mk_A(v_1, v_2, ..., v_n)

The make-function yields a composite value of composite type A whose field values are v_i (i =1,…,n) that corresponds to fields f_1, f_2, ..., f_n, respectively. For example,

account = mk_Account(1073548, 1234, 5000)

(2) Operators

(2.1) Field select

Let co be a variable of composite type A.
Then, we use

co.f_i

to represent the field f_i (i =1,...,n) of the composite object co.

Examples:

account.password
account.balance

(2.2) Field modification (modify)

Given a composite value, say co, of type A, we can apply the field modification operator modify to create another composite value of the same type.

modify(co, f_1 -> v_1, f_2 -> v_2, ..., f_n -> v_n)

Example:
let account = mk_Account(1073548, 1234, 5000)
Then, we can have the expression:

account1 = modify(account, password -> 4321)

(3) Comparison

Two composite values can be compared to determine whether they are identical or not.

Examples:

mk_Account(1073548, 1234, 5000) =
  mk_Account(1073548, 1234, 5000)

mk_Account(1073548, 1234, 5000) <>
  mk_Account(1073548, 4321, 5000)

Product types

A product type defines a set of tuples with a fixed length.

Let T_1, T_2, ..., T_n be n types.
Then, a product type T is declared as follows:

T = T_1 * T_2 * ... * T_n

A value of T is created using the make-function:

mk_T(v_1, v_2, ..., v_n)

Example:

Suppose type Date is declared as:

Date = nat0 * nat0 * nat0

Then

mk_Date(1999, 7, 25)

mk_Date(2000, 8, 30)

mk_Date(2001, 7, 10)

are the values of type Date.

Examples: the use of tuples:

```
d: Date;

d = mk_Date(1999, 7, 25)

d = mk_Date(2000, 8, 30)

d = mk_Date(2001, 7, 10)
```

There are two operators on tuples: tuple application and tuple modification.

(1) A tuple application yields an element of the given position in the tuple, whose general format is:

a(i): T * nat --> T_i

where a is a variable of product type T; i is a natural number indicating the position of the element referred to in tuple a; and T_i denotes the ith type in the declaration of T.

For example, let

date1 = mk_Date(1999, 7, 25)

date2 = mk_Date(2000, 8, 30)

Then, the following results can be derived:

date1(1) = 1999

date1(2) = 7

date1(3) = 25

date2(1) = 2000

date2(2) = 8

date2(3) = 30

A tuple can also be directly used in applications.

Examples:

mk_Date(2000, 8, 30)(2) = 8

mk_Date(2000, 8, 30)(3) = 30

(2) A tuple modification is similar to a composite value modification. The same operator modify is also used for tuple modification, but with slightly different syntax:

```
modify: T * T_1 * T_2 * ... * T_n -> T
modify(tv, 1 -> v_1, 2 -> v_2, ..., n -> v_n)
```

where T is a product type, T_i (i =1,…,n, n >= 1) are the element types. This operation yields a tuple of the same type based on the given tuple tv, with the first element being v_1, the second element being v_2, and so on.

Examples:

```
modify(mk_Date(2000, 8, 30), 1 --> 2001, 3 --> 20) =
    mk_Date(2001, 8, 20)

modify(mk_Date(2001, 8, 20), 2 --> 15) =
    mk_Date(2001, 15, 20)
```

An example of specification

Suppose we want to build a table to record students' credits resulting from two courses:

personal data	course1	course2	total
Helen,0001,A3	2	2	4
John, 0002,A2	0	2	2
...	...	...	...

We aim to build several processes on this kind of table.

This table can be perceived as a sequence of student data. That is,

T = [OneStudent1, OneStudent2, …,
OneStudentn]

```
module Students_Record;
 type
   CourseCredit = nat0;
   TotalCredit = nat0;
   PersonalData = composed of
                    name: string
                    id: nat0
                    class: string
                  end;
  OneStudent = PersonalData * CourseCredit * CourseCredit *
                TotalCredit;
  StudentsTable = seq of OneStudent;

 var
   students_table: StudentsTable;

 inv
  forall[i, j: inds(students_table)] | i <> j => students_table(i)(1).id <>
                                                  students_table(j)(1).id);
```

```
process Search(search_id: nat0)
                        info: OneStudent
ext rd students_table
pre exists[i: inds(students_table)] |
   students_table(i)(1).id = search_id
post exists[i: inds(students_table)] |
   students_table(i)(1).id = search_id and
   info = students_table(i)]
end_process;
```

```
process Update(one_student: OneStudent, credit1,
                        credit2: CourseCredit)
ext wr students_table
pre  exists[i: inds(students_table)] |
                              students_table(i) = one_student
post len(students_table) = len(~students_table) and
       forall[i: inds(~students_table)] |
          (~students_table(i) = one_student =>
           students_table(i) =
              modify(~students_table(i), 2 -> credit1,
                                    3 -> credit2,
                                    4 -> credit1 + credit2)) and
    (~students_table(i) <> one_student =>
    students_table(i) = ~students_table(i))
end_process;
end_module;
```

Class exercise 7

1. Let a = mk_Account(010, 300, 5000), where the type Account is defined as follows:

```
Account = composed of
          account_no: nat1
          password: nat1
          balance: real
          end
```

Then evaluate the expressions:

a. a.account_no = ?
b. a.password = ?
c. a.balance = ?
d. modify(a, password -> 250) = ?
e. modify(mk_Account(020, 350, 4050), account_no -> 100,
   balance -> 6000) = ?

3. Let x be a variable of the type Date defined as follows:

Date = nat0 * nat0 * nat0

Let x = mk_Date(2002, 2, 6).

Then evaluate the expressions:

a. x(1) = ?

b. x(2) = ?

c. x(3) = ?

d. modify(x, 1 -> 2003)

e. modify(x, 2 -> 5, 3 -> 29)

f. modify(x, 1 -> x(1), 2 -> x(2))

4. Define a composite type Student that has the fields: name, date_of_birth, college, and grade. Write specifications for the processes: Register, Change_Name, Get_Info. The Register takes a value of Student and adds it to the external variable student_list, which is a sequence of students. Change_Name updates the name of a given student with a new name in student_list. Get_Info provides all the available field values of a given student in student_list.