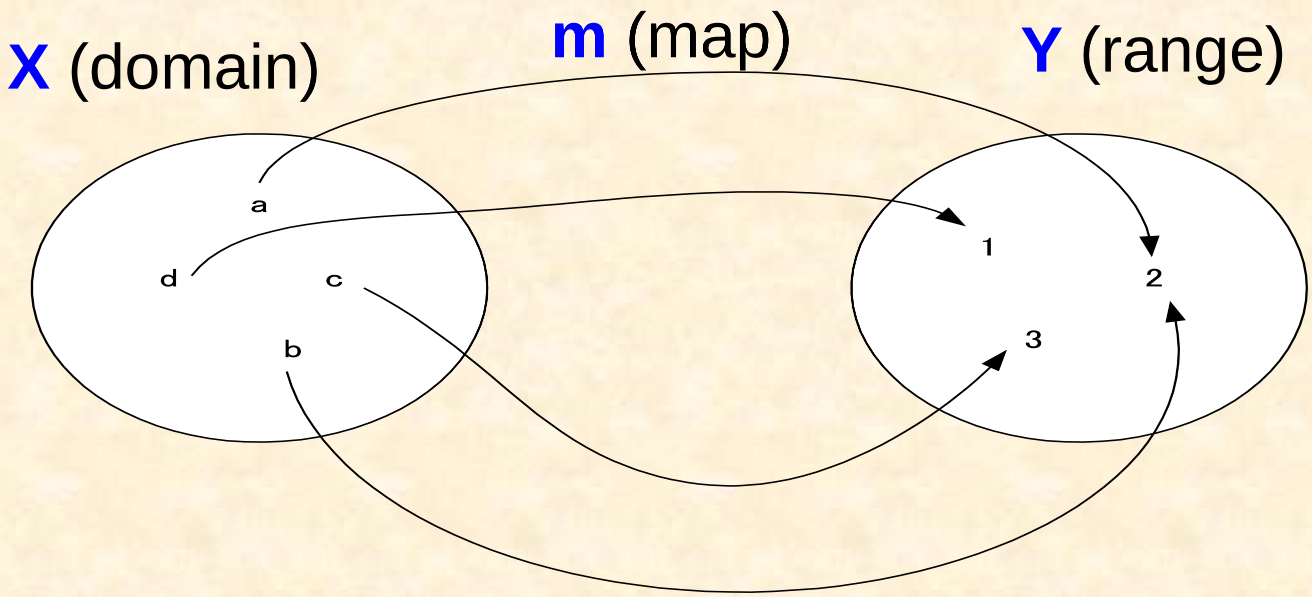
Map types

Contents:

- What is a map?
- The type constructor
- Operators
- Specification using maps

What is a map?

A map is a finite set of pairs, describing an association between two sets. It is a special function.



forall[x1, x2: X] | x1 <> x2 and m(x1) inset Y and m(x2) inset Y => m(x1) <> m(x2)

A map (or sometimes we call it "map value") is represented with a notation similar to the set notation:

{a_1 -> b_1, a_2 -> b_2, ..., a_n -> b_n}

Each a_i -> b_i (i =1,...,n) denotes a pair which is known as maplet.

For example, the map illustrated in the Figure on the previous slide is expressed as follows:

{a -> 2, b -> 2, c -> 3, d -> 1}

An empty map is expressed as:

{->}

Important property:

A map usually describes a many-to-one association: it allows the mapping from many elements in the domain to the same element in the range, but does not allow the mapping from the same element in the domain to different elements in the range.

The type declaration

A map type T is declared based on the domain type T1 and the range type T2 in the following format:

T = map T1 to T2

T contains all the possible maps that associate values in T1 with the values in T2.

Another example:

A = map nat to char

declares a map type A whose domain type is nat and range type is char.

Examples: possible maps (or map values) of type A:

{1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd'}
{5 -> 'u', 15 -> 'v', 25 -> 'w'}
{10 -> 'x', 20 -> 'y'}
{50 -> 'r'}
{->}

Note that: domain type and range type of a map type can be an infinite set, although a concrete map value derived from the map type must contain only finite maplets (elements of a map).

Operators

(1) Constructors

Two constructors: map enumeration and
map comprehension.

(1.1) Map enumeration

The general format:
{a_1 -> b_1, a_2 -> b_2, ..., a_n -> b_n}

Examples:
{3 -> 'a', 8 -> 'b', 10 -> 'c'}

{“Beijing Jiaotong University" -> “China“, "Hosei University" -> "Japan", "University of Manchester" -> "U.K."}

{1 -> s(1), 2 -> s(2), 3 -> s(3)}

(1.2) Map comprehension

{a -> b | a: T1, b: T2 & P(a, b)} or

{a -> b | P(a, b)}

Example:

{x -> y | x: {5, 10, 15}, y: {10, 20, 30} & y = 2 * x} =
{5 -> 10, 10 -> 20, 15 -> 30}

defines a map.

The following map comprehension defines an illegal map:

{x -> y | x: {1, 2, 3}, y: {5, 10, 15, 20} & y > x * 5} =
{1 -> 10, 1 -> 15, 1 ->20, 2 -> 15, 2 -> 20, 3 -> 20}

Why?

(2) Other operators

(2.1) Map application

Let m be a map:

m: map T1 to T2;

Then, m can be applied to an element in its domain to yield an element in its range.

Example:

m(a)

denotes an application to element a in its domain.

Example: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30}

Then

m1(5) = 10

m1(10) = 20

m1(15) = 30

Note that when m1 applies to number 2, for example, the result of the application is undefined:

m1(2) = undefined.

(2.2) Domain and range (dom, rng)

Let m be a map:

m: map T1 to T2;

Then, the domain of m is a subset of T1 and its range is a subset of T2, which can be obtained by applying the operators dom and rng, respectively.

dom: map T1 to T2 --> set of T1

Example:

let m1 = {5 -> 10, 10 -> 20, 15 -> 30}

Then

dom(m1) = {5, 10, 15}

The range operator rng yields, when applied to a map, the set of the second elements of all the maplets in the map.

rng: map T1 to T2 --> set of T2

rng(m) == {m(a) | a inset dom(m)}

Example: let m1 = {5 -> 10, 10 -> 20, 15 -> 30}.

Then,

rng(m1) = {10, 20, 30}

(2.3) Domain and range restriction to (domrt, rngrt)

Given a map and a set, sometimes we may want to obtain the submap of the map whose domain or range is restricted to the set. Such operations are known as domain restriction to and range restriction to, respectively.

domrt: set of T1 * map T1 to T2 --> map T1 to T2
domrt(s, m) == {a -> m(a) | a inset inter(s, dom(m))}

rngrt: map T1 to T2 * set of T2 --> map T1 to T2 rngrt(m, s)
== {a -> m(a) | m(a) inset inter(s, rng(m))}

Examples: let

$m1 = \{5 \rightarrow 10, 10 \rightarrow 20, 15 \rightarrow 30\}$

$s1 = \{5, 10\}$.

Then,

$domrt(s1, m1) = \{5 \rightarrow 10, 10 \rightarrow 20\}$

$rngrt(m1, s1) = \{5 \rightarrow 10\}$

(2.4) Domain and range restriction by (domrb, rngrb)

In contrast to "domain restriction to" and "range restriction to" operations, sometimes we may want to derive a submap of a map whose domain or range is the subset of the domain or range of the map that is disjointed with a given set. Such operations are called domain restriction by and range restriction by, respectively.

```
domrb: set of T1 * map T1 to T2 --> map T1 to T2
domrb(s, m) == {a -> m(a) | a inset diff(dom(m), s)}

rngrb: map T1 to T2 * set of T2 --> map T1 to T2
rngrb(m, s) == {a -> m(a) | m(a) inset diff(rng(m), s)}
```

Examples: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30}

s1 = {5, 10}.

Then,

domrb(s1, m1) = {15 -> 30}

rngrb(m1, s1) = {10 -> 20, 15 -> 30}

(2.5) Override (override)

Overriding is an operation for a union of two maps m1 and m2, denoted by override(m1, m2), with the restriction: if a maplet in map m2 shares the first element with a maplet in m1, the resulting map only includes the maplet in m2 as its element.

```
override: map T1 to T2 * map T1 to T2 -->
          map T1 to T2
override(m1, m2) == {a -> b |
    a: union(dom(m1), dom(m2)) &
    a inset dom(m2) => b = m2(a) and
    a notin dom(m2) => b = m1(a)}
```

Example: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30},

m2 = {10 -> 5, 15 -> 50, 4 -> 20}.

Then,

override(m1, m2) =

{10 -> 5, 15 -> 50, 4 -> 20, 5 -> 10}

Notice: override is not commutative, that is,
override(m1, m2) <> override(m2, m1)

holds in general.

Example: compare override(m1, m2) to the following:

override(m2, m1) = {5 -> 10, 10 -> 20,
15 -> 30, 4 -> 20}

(2.6) Map inverse (inverse)

Map inverse is an operation that yields a map from a given map by exchanging the first and second elements of every maplet of the given map.

```
inverse: map T1 to T2 --> map T2 to T1
inverse(m) == {a --> b | a: rng(m), b: dom(m)
                        & a = m(b)}
```

Example: let m1 = {5 -> 10, 8 -> 20, 2 -> 30}

Then,

inverse(m1) = {10 -> 5, 20 -> 8, 30 -> 2}

However, if the map defines a many-to-one rather than one-to-one association between its domain and range, the application of the inverse operator is undefined.

(2.7) Map composition (comp)

Map composition is an operation that forms a another map from two given maps.

```
comp: map T1 to T2 * map T2 to T3 -->
          map T1 to T3
comp(m1, m2) == {a -> b | a: dom(m1),
                          b: rng(m2) &
                 exists[x: rng(m1)] |
                   x inset dom(m2) and
                   x = m1(a) and b = m2(x)}
```

Example: let

m1 = {5 -> 10, 8 -> 20, 2 -> 4},

m2 = {10 -> 5, 15 -> 5, 4 -> 20},

Then, the composition of m1 and m2 is:

comp(m1, m2) = {5 -> 5, 2 -> 20}

(2.8) Equality and inequality (=, <>)

We use m1 = m2 to mean m1 is identical to m2, and m1 <> m2 to mean m1 is different from m2. Formally,

```
m1 = m2 <=> dom(m1) = dom(m2) and
                rng(m1) = rng(m2) and
                forall[a: dom(m1), b: rng(m1)] |
                      b = m1(a) <=> b = m2(a)

m1 <> m2 <=> not m1 = m2
```

Specification using maps

Let us reconsider defining the type Account with a map type. Since every customer's account number is unique and it is common to allow one customer to have only one account of the same kind in the same bank, the customer account can be modeled as a map from the account number to the account data including password and balance.

Account = map AccountNumber to AccountData;

AccountNumber = nat;

AccountData = composed of

password: nat

balance: real

end;

We then redefine the processes

Check_Password, Withdraw, and Show_Balance

as follows:

```
process Check_Password(card_id: AccountNumber, pass: nat)
                                        confirm: bool
ext rd account_file: Account
post card_id inset dom(account_file) and
       account_file(card_id).password =  pass and
       confirm =  true
       or
       card_id notin dom(account_file) and
       confirm = false
comment
    If the given account number card_id and password pass
    matches with the account_file, the output confirm becomes
    true; otherwise, it becomes false.
end_process;
```

```
process Withdraw(card_id: AccountNumber, amount: real)
                              cash: real
 ext wr account_file: Account
 pre card_id inset dom(account_file) and amount <=
    account_file(card_id).balance
 post account_file = override(~account_file, {card_id ->
        mk_AccountData(~account_file(card_id).password,
             ~account_file(card_id).balance - amount)} and
    cash = amount
comment
   The precondition requires that the provided card_id be
    registered in the account_file and the requested amount to
    withdraw be less than or equal to the current balance. The
    updating of the current balance of the account with the
    account number card_id is expressed by a map overriding
    operation: the updated balance is the result of subtracting
    the requested amount from the current balance.
end_process;
```

```
process Show_Balance(card_id:
                         AccountNumber) bal: real
ext rd account_file: Account
pre card_id inset dom(account_file)
post bal = account_file(card_id).balance
comment
  The account number card_id must exist in the
   account_file before the execution of the
   process. The assignment of the current
   balance to the output variable bal is reflected
   by a map application in the postcondition.
end_process;
```

Class exercise 8

1. Let m1 and m2 be two maps of the map type from nat0 to nat0;

   m1 = {1 -> 10, 2 -> 3, 3 -> 30},

   m2 = {2 -> 40, 3 -> 1, 4 -> 80}, and s = {1, 3}.

Then, evaluate the expressions:

a. dom(m1) = ?

b. dom(m2) = ?

c. rng(m1) = ?

d. rng(m2) = ?

e. domrt(s, m1) = ?

f. domrt(s, m2) = ?

g. rngrt(m1, s) = ?

h. rngrt(m2, s) = ?

i. domrb(s, m1) = ?

j. domrb(s, m2) = ?

k. rngrb(m1, s) = ?

l. rngrb(m2, s) = ?

m. override(m1, m2) = ?

n. override(m2, m1) = ?

o. inverse(m1) = ?

p. inverse(m2) = ?

q. comp(m1, m2) = ?

r. comp(m2, m1) = ?

s. m1 = m2 <=> ?

t. m1 <> m2 <=> ?

2. Define BirthdayBook as a map type from the type Person (with the fields: id, name, and age) to the type Birthday, and specify the processes: Register, Find, Delete, and Update. All the processes access or update the external variable birthday_book of the type BirthdayBook. The process Register adds a person's birthday to birthday_book. Find detects the birthday for a person in birthday_book. Delete eliminates the birthday of a person from birthday_book. Update replaces the wrong birthday registered in birthday_book with a correct birthday.