

Modelling using Mappings

- Mappings:
 - The finite mapping type constructor
 - Value definitions: enumeration, comprehension
 - Operators on mappings
- Case Study: the tracking manager

The finite mapping type constructor

A mapping is a functional relationship between two sets of values: a domain and a range. Mappings are common in many models, e.g.

"Each bank account has exactly one balance"

"Each reactor has an input, an output and an operating temperature."

Mappings represent one-to-one or many-to-one relationships, *but not one-to-many!*

The finite mapping type constructor

The mapping type constructor is `map X to Y`

where `X` and `Y` are data types

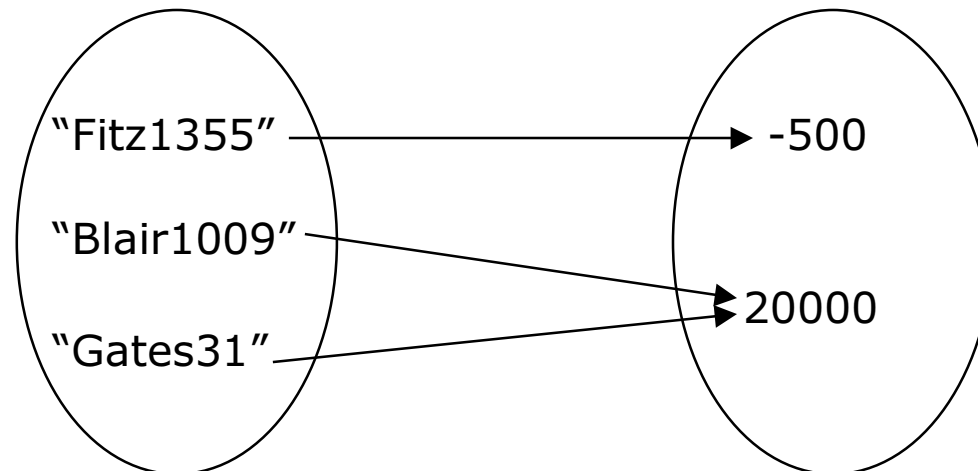
e.g. "each bank account has exactly one bank balance":

`AccountNumber = seq of char`

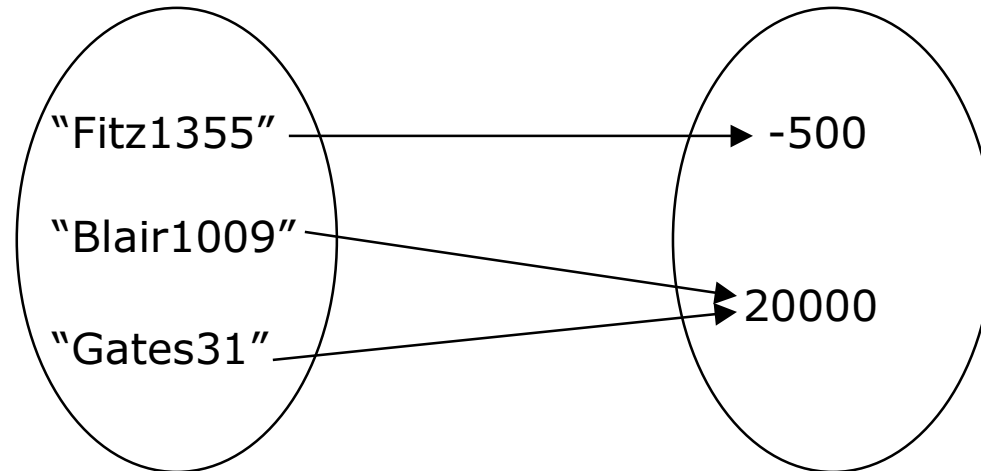
`Balance = int`

`Accounts = map AccountNumber to Balance`

An example
mapping:



Value definitions: enumeration, comprehension



To enumerate a mapping, we present the related domain element-range element pairs (called *maplets*). For the mapping illustrated above, the enumeration would be:

A maplet relating domain element x to range element y is written

$x \mid \rightarrow y$

Value definitions: enumeration, comprehension

Enumerate the mapping that relates natural numbers less than 4 to their factorials:

We could express the same mapping as a comprehension (use the auxiliary function Fac to calculate the factorial).

Exercise: define the function Fac .

Value definitions: enumeration, comprehension

A mapping comprehension has the following form:

$$\{ \textit{expression} \mid \rightarrow \textit{expression} \mid \textit{binding} \ \& \ \textit{predicate} \}$$

The mapping consisting of the maplets formed by evaluating the expressions under each assignment of values to bound variables satisfying the predicate.

Consider all the values that can be taken by the variables in the binding.

Restrict this to just those combinations of values which satisfy the predicate.

Evaluate the expressions for each combination. This gives you the maplets in the mapping.

e.g. $\{ x \mid \rightarrow x/2 \mid x:\text{nat} \ \& \ x < 5 \}$

Value definitions: enumeration, comprehension

Like sets and sequences, mappings are finite. Are the following mappings defined?

$$\{ x \mapsto x^2 \mid x:\text{nat1} \ \& \ x^2 > 3 \}$$
$$\{ x \mapsto y \mid x,y:\text{nat1} \ \& \ x < 4 \ \text{and} \ y < 3 \}$$
$$\{ x \mapsto x^2 \mid x:\text{int} \ \& \ x < 10 \}$$

Operators on mappings

dom: map A to B \rightarrow set of A

Domain

rng: map A to B \rightarrow set of A

Range

Evaluate the following:

dom { n \mid \rightarrow 3*n | n:nat & n<50 }

rng { n \mid \rightarrow 3*n | n:nat & n<50 }

Operators on mappings

$_(_) : \text{map } A \text{ to } B * A \rightarrow B$

Mapping Lookup

For a mapping m and a domain element a , the expression

$m(a)$

denotes the range element pointed to by a .

Is this a total or a partial operator?

Operators on mappings

Example of mapping lookup:

```
Accounts = map AccountNumber to Balance
```

Define a function with the following signature which returns the names of overdrawn account holders:

```
overdrawn : Accounts -> set of AccountNumber
```

Operators on mappings

The mapping merge or mapping union operator joins two mappings together:

```
_ munion _ : (map A to B) * (map A to B) -> (map A to B)
```

Example:

```
{ "John" |-> -500, "Tony" |-> 20000 }  
munion { "Cherie" |-> 150 }  
= { "John" |-> -500, "Tony" |-> 20000, "Cherie" |-> 150 }
```

This operator is partial. Can you see why?

Operators on mappings

Mapping union is only defined on inputs that are *compatible*. We can define a function to check for mapping compatibility:

```
compatible: (map A to B) * (map A to B) -> bool
```

```
compatible(m1,m2) ==
```

```
forall x in set dom m1 inter dom m2 & m1(x) = m2(x)
```

Operators on mappings

An alternative operator is the mapping override operator:

```
_ ++ _ : (map A to B) * (map A to B) -> (map A to B)
```

This operator is defined just like `munion`, except that where `m1` and `m2` are not compatible, `m2` wins, e.g.

```
{ "John" |-> -500, "Tony" |-> 20000 }  
++ { "Tony" |-> 300, "Cherie" |-> 150 }  
= { "John" |-> -500, "Tony" |-> 300, "Cherie" |-> 150 }
```

A very common use of this operator is to update a mapping at a point, e.g.

```
m ++ {x |-> e}
```

updates the mapping `m` so that `x` now points to `e`.

Operators on mappings

There are some operators to modify mappings by restricting the domain or range:

$$_ \leftarrow : _ : (\text{set of } A) * (\text{map } A \text{ to } B) \rightarrow (\text{map } A \text{ to } B)$$

The expression $s \leftarrow : m$ is the same as m except that the elements of s have been removed from its domain (and any unattached range elements are removed too).

$$_ < : _ : (\text{set of } A) * (\text{map } A \text{ to } B) \rightarrow (\text{map } A \text{ to } B)$$

The expression $s < : m$ is the same as m except that the domain is restricted down to just the elements of s (and any unattached range elements are removed too).

$$_ \rightarrow : _ : (\text{map } A \text{ to } B) * (\text{set of } B) \rightarrow (\text{map } A \text{ to } B)$$

The expression $m \rightarrow : s$ is the same as m except that the elements of s have been removed from its range (and any unattached domain elements are removed too).

$$_ :> _ : (\text{map } A \text{ to } B) * (\text{set of } B) \rightarrow (\text{map } A \text{ to } B)$$

The expression $m :> s$ is the same as m except that the range is restricted down to just the elements of s (and any unattached domain elements are removed too).

Details are in the text.

Operators on mappings

Example:

Define a function returning the accounts mapping for those account holders who are not overdrawn.

```
AccountNumber = seq of char
```

```
Balance = int
```

```
Accounts = map AccountNumber to Balance
```

```
credit-map : Accounts -> Accounts
```

```
credit-map (acs) ==
```

The tracking manager example

A model of an architecture for tracking the movement of containers of hazardous waste as they go through reprocessing was developed by a team in Manchester Informatics with BNFL (Engineering) in 1995.

The **purpose** of the model was to establish the rules governing the movement of containers of waste which the tracking manager would have to enforce. The model was safety-related, but note that the model was built simply in order to understand the problem better, not as a basis for software development. *Models don't just have to serve as specifications.*

The tracking manager

Basic data types

At the top level, the tracker holds information about containers and the phases of the plant:

```
Tracker :: containers : ContainerInfo
         phases      : PhaseInfo
```

The container and phase information is modelled as a mapping from identifiers to details (*this is a very common use of mappings, with identifiers in the domain and data types defining details in the range*)

```
ContainerInfo = map ContainerId to Container
```

```
PhaseInfo = map PhaseId to Phase
```

The details of how identifiers are represented are immaterial:

ContainerId =

PhaseId =

For each container, we record the fissile mass of its contents and the kind of material it contains.

Container

Material

The tracking manager

Basic data types

Each phase houses a number of containers, expects certain material types and has a maximum capacity.

Try modelling this yourself:

The tracking manager

Basic data types

In the real tracking manager project, domain experts from BNFL were closely involved with the development of the formal model. We relied on the domain experts to point out the safety properties that had to be respected by the tracker. For example, the number of containers in a phase should not exceed the phase's capacity:

```
Phase :: contents : set of ContainerId
       capacity  : nat
```

```
inv p == card p.contents <= card p.capacity
```

The domain experts from BNFL often commented that this ability to record constraints formally as invariants was extremely valuable.

The tracking manager

The tracker invariant

```
Tracker :: containers : ContainerInfo  
        phases      : PhaseInfo
```

```
inv mk_Tracker(containers, phases) ==
```

1. all of the containers present in phases are known about in the containers mapping.
2. no two distinct phases may have any containers in common.

```
Tracker :: containers : ContainerInfo  
        phases      : PhaseInfo
```

```
inv mk_Tracker(containers, phases) ==
```

```
    Consistent(containers, phases) and  
    PhasesDistinguished(phases)
```

The tracking manager

The tracker invariant

```
Consistent: ContainerInfo * PhaseInfo -> bool
```

```
Consistent(containers, phases) ==
```

```
-- all of the containers present in phases are known
```

```
-- about in the containers mapping.
```

```
forall ph in set rng phases &
```

```
  ph.contents subset dom containers
```

The tracking manager

The tracker invariant

```
PhasesDistinguished: PhaseInfo -> bool
PhaseDistinguished(phases) ==
  -- no two distinct phases may have any containers
  -- in common
  not exists p1, p2 in set dom phases &
    p1 <> p2 and
    phases(p1).contents inter phases(p2).contents <> {}
```

The tracking manager

Tracker functionality

- introduce a new container to the tracker, giving its identifier and contents;
- give permission for a container to move into a given phase;
- remove a container from a phase;
- delete a container from the plant.

```
Introduce: Tracker * ContainerId * real * Material  
          -> Tracker
```

```
Introduce(trk, cid, quan, mat) ==  
  mk_Tracker(trk.containers munion  
             {cid |-> mk_Container(quan,mat)},  
             trk.phases)
```

```
pre cid not in set dom trk.containers
```

The tracking manager

Tracker functionality

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers, phases), cid, dest) ==
  -- must check that the tracker invariant will be
  -- maintained by the move
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity
```

The tracking manager

Tracker functionality

```
Remove: Tracker * Containerid * PhaseId -> Tracker
```

```
Remove(mk_Tracker(containers, phases), cid, pid) ==
```

```
mk_Tracker(containers,
```

```
    phases ++ {pid | ->
```



```
pre pid in set dom phases and
```

```
    cid in set phases(pid).contents
```

The tracking manager

Tracker functionality

We can simplify function definitions by using a local declaration given in a **let** expression:

```
Remove: Tracker * Containerid * PhaseId -> Tracker
Remove(mk_Tracker(containers, phases), cid, pid) ==
let pha = mk_Phase(phases(pid).contents \ {cid},
                    phases(pid).capacity)
in
mk_Tracker(containers, phases ++ {pid |-> pha})
pre pid in set dom phases and
    cid in set phases(pid).contents
```

The tracking manager

Tracker functionality

To delete a container, two things have to be done:

- we have to remove the container from the `containers` mapping; and
- we have to remove the container from the phase in which it occurs (just as in the `Remove` function).

```
Delete: Tracker * Containerid * PhaseId -> Tracker
```

```
Delete(tkr, cid, pid) ==
```

```
mk_Tracker({cid} <-: containers,
```

```
          Remove(tkr, cid, pid).phases)
```

```
pre pre_Remove(tkr, cid, source)
```

Review

- Mappings represent functional relations between two sets of values: the domain and the range.
- Mapping values can be defined by enumeration or comprehension as sets of maplets.
- Operators to extract the domain and range, mapping lookup and for combining mappings by union or override. Some operators are partial.