Abstract Specification

First we must define (show) relevant parts of the abstract specification.

Basic data types: $[NAME, ID, INT]$.

Schemas that capture the system state:

Bank

$$persons : NAME \pfun Person$$
$$accounts : ID \pfun Account$$
$$ownership : Person \rel Account$$

Account

$$owner : Person$$
$$id : ID$$
$$balance : INT$$

$$balance \geqslant 0$$

We do not need the structure of the data type *Person* here.

Abstract schema for the operation *CreateAccount*.

CreateAccount

$$\Delta Bank$$
$$per? : Person$$
$$nid? : ID$$
$$acc! : Account$$

$$nid? \notin \dom accounts$$
$$ownership' = ownership \oplus \{(per?, acc!)\}$$
$$acc!.owner = per? \land acc!.id = nid?$$

The Z language supports the dot-notation for accessing fields.

Concrete Design

We can use arrays in the concrete version of the schema *Bank*.

$arrPersons$: `array`$[1..N]$ `of` $PersonImpl1$
$arrAccounts$: `array`$[1..N]$ `of` $AccountImpl1$

Arrays may be defined as explicitly finite (1..N) or as infinite with constraints on their size expressed through logic formulas.

The digit 1 in the name of the concrete schemas (*PersonImpl*1, *AccountImpl*1) indicates that it is the first step of iterative refinement. We will use a similar notation in the whole document.

For the purpose of mathematical reasoning about the concrete data structures, we can model arrays as functions from positive integers ($\mathbb{N}_1$) to element objects (of type *PersonImpl*1, respectively *AccountImpl*1).

$arrPersons : \mathbb{N}_1 \to PersonImpl1$
$arrPccounts : \mathbb{N}_1 \to AccountImpl1$

An array element $persons[i]$ is simply the function value $persons(i)$. The assignment $persons[i] := p$ is exactly specified as $persons' = persons \oplus i \mapsto p$.

Schema for the concrete state is actually very similar to the abstract schema. We just need some variables to capture the current array sizes.

*BankImpl*1

$$arrPersons : \mathbb{N}_1 \to PersonImpl1$$
$$arrAccounts : \mathbb{N}_1 \to AccountImpl1$$
$$szPersons : \mathbb{N}$$
$$szAccounts : \mathbb{N}$$

$$\forall i, j : 1..szPersons \bullet i \neq j \Rightarrow arrPersons(i) \neq arrPersons(j)$$

The constraint expresses our requirement that we want unique persons (distinct elements of the array).

Note that we use different names of state variables in the concrete schema (*BankImpl*1) than in the abstract schema to avoid confusion.

We also need to define the *abstraction schema* that captures the relationship between abstract states (schema *Bank*) and concrete states (schema *BankImpl*1).

AbsBank

$$Bank$$
$$BankImpl1$$

$$(p, a) \in ownership \Leftrightarrow (\exists i : 1..szPersons \bullet p = arrPersons(i)) \land (\exists j : 1..szAccounts \bullet a = arrAccounts(j)) \land (a.owner = p)$$

The abstraction schema *AbsBank* includes both the abstract and concrete schema, and then it defines some relation between abstract and concrete state variables. In literature, you can also find the term *abstraction relation* for this concept.

To have a really complete design, we would have to define also some concrete representation of the relation *ownership*, and use it in the abstraction schema, but I did not want to make the example too complex for illustration purposes.

The concrete schema (implementation) of the operation *CreateAccount* may look like this:

*CreateAccountImpl*1

$$\Delta BankImpl1$$
$$per? : PersonImpl1$$
$$nid? : ID$$
$$acc! : AccountImpl1$$

$$\exists i : 1..szPersons \bullet per? = arrPersons(i)$$

$$\forall j : 1..szAccounts \bullet nid? \neq arrAccounts(j).id$$

$$szAccounts' = szAccounts + 1$$

$$arrAccounts' = arrAccounts \oplus \{szAccounts' \mapsto acc!\}$$

$$acc!.owner = per? \land acc!.id = nid?$$

The concrete schema has the same arguments (inputs) and outputs as the abstract schema, but operates on the concrete state (data structures). Note that we again omitted constraints involving the relation *ownership* to avoid unnecessary complexity of this example.