Why3: State-based Development

A deductive verification tool like Why3 can be used to check the behavior of a collection of functions working on a shared state, such as classes in object-oriented programming (sharing instance variables), smart contracts in blockchain development, or multi-threaded concurrent programming (based on shared memory).

We will illustrate this by means of an example where we model functions operating on a bank account.
The global shared state stores the balance of each account, identified by a number.

```
module Accounts_MapImp
  use int.Int
  type accNumber = int
  type amount    = int

  val open (n :accNumber) : ()
  val deposit (n :accNumber) (x :amount) : ()
  val withdraw (n :accNumber) (x :amount) : ()
  val transfer (from :accNumber) (to_ :accNumber) (x :amount)
: ()
```

We need a way to associate balances with account numbers. We will use a Why3 **finite map** type for this purpose. Map types implement dictionaries; the Why3 library https://why3.lri.fr/stdlib/fmap.html makes the following available:

- a logic-level map type (module `Fmap`) that cannot be extracted to code;
- and two programming map types, that can be extracted (typically implemented as hash tables)
  - a "functional" type (module `MapApp`)
  - and an "imperative" type (module `MapImp`)

Since we want to program with maps we will use programming types.

We start by using the imperative type. The read and write functions on this type are specified in the library as follows. Note that the `find` and `add` functions used in the contracts are synonymous logic-level functions: the behavior of programming-level maps is specified using the `Fmap` logic type.

```
  val find (k: key) (m: t 'v) : 'v
    requires { mem k m }
    ensures  { result = m[k] }
    ensures  { result = find k m }

  val add (k: key) (v: 'v) (m: t 'v) : unit
    writes  { m }
    ensures { m = add k v (old m) }
```

Programming types are used by *cloning* the respective module into our own. In this concrete case we will instantiate the type of the map keys:

```
clone fmap.MapImp with type key = accNumber
```

The type of maps with `accNumber` as keys is now available with name `t`. The following declares the state variable `accounts` as a map with values of type `amount`:

```
  val accounts : t amount
```

We wish to write functions to do the following tasks:

- *open* a new account with a given account number
- *deposit* funds into a given account
- *withdraw* funds from an account
- *transfer* funds internally from an account to another

As a first step let us consider a natural language specification of their behavior. For instance:

- given an account number `n` , the call `open n` will insert the pair (`n` →0) in the `accounts` dictionary

- given an account number `n` and an amount `x`, the call `deposit n x` will add `x` to the current balance of account `n`

The next step is to discuss what the functions should do in case they receive unexpected argument values. What should happen if

- in a call `open n`, the account number `n` already exists, i.e. it is already present in the domain of `accounts` ?

- in a call `deposit n x`,
  - the number `n` is not a valid account number (i.e. it is not in the domain of `accounts`,
  - or `x` is a negative number ?

We will choose *not to program defensively* : the functions will take for granted that the above situations do not occur.

So the definitions are really simple:

```
let open (n :accNumber) : ()
= add n 0 accounts

let deposit (n :accNumber) (x :amount) : ()
= let bal = find n accounts in
  add n (bal+x) accounts
```

Now, in the absence of defensive programming it is the caller's responsibility to make sure that all calls satisfy the desired conditions, which should be included in the contracts of the callee functions as preconditions:

```
  let open (n :accNumber) : ()
    requires { not mem n accounts }
  = add n 0 accounts

  let deposit (n :accNumber) (x :amount) : ()
    requires { mem n accounts /\ x > 0 }
  = let bal = find n accounts in
    add n (bal+x) accounts
```

We will now also include postconditions in the functions' contracts. This will not only allow us to prove that the implementations respect the specifications (expressed above in natural language), but also that no calls are made that do not respect the preconditions.

```
  let open (n :accNumber) : ()
    requires { not mem n accounts }
    ensures  { mem n accounts /\ find n accounts = 0 }
    ensures  { forall a :accNumber. mem a accounts <-> mem a (
old accounts) \/ a = n }
    writes   { accounts }
  = add n 0 accounts

  let deposit (n :accNumber) (x :amount) : ()
    requires { mem n accounts /\ x > 0 }
    ensures  { find n accounts = find n (old accounts) + x }
    ensures  { forall a :accNumber. mem a accounts  /\ a <> n
```

```
                    -> find a accounts = find a (old accounts)
}
    ensures  { forall a :accNumber. mem a accounts <-> mem a
(old accounts) }
    writes   { accounts }
  = let bal = find n accounts in
    add n (bal+x) accounts
```

Note that:

- The postcondition highlighted in `deposit` expresses an obvious fact that is implicit in the natural language spec, but should be stated explicitly in the contract: all balances are preserved, with the exception of account `n`
- The contracts also include postconditions relating the keysets (domains) of the mapping before and after execution of each function
- The frame conditions `writes ...` make explicit the *effects* of the functions, i.e. the parts of the global state that are modified by them

State Invariants

We may wish to prove that certain properties of the global state always hold, i.e. they are invariants of all the state-changing functions.

For instance:

> *The balance of every account is non-negative*

Such properties may be treated by simply including them simultaneously and pre- and postconditions in all functions. We may then add the following to the contracts of the functions defined above:

```
    requires { forall a :accNumber. mem a accounts -> find a a
ccounts >= 0 }
    ensures  { forall a :accNumber. mem a accounts -> find a a
ccounts >= 0 }
```

Exercise
Complete the definition of the module by equipping the remaining functions with appropriate contracts and proving their correctness.

```
  let withdraw (n :accNumber) (x :amount) : ()
  = let bal = find n accounts in
    add n (bal-x) accounts

  let transfer (from :accNumber) (to_ :accNumber) (x :amount)
: ()
  = let balfrom = find from accounts in
    let balto   = find to_  accounts in
    add to_  (balto  +x) accounts ;
    add from (balfrom-x) accounts
```

Record types and type invariants

We could alternatively used a declarative / functional type for maps. We will illustrate their use with an alternative implementation of the above module

```
module Accounts_MapApp_Record
  use int.Int
```

```
  type accNumber = int
  type amount    = int
  clone fmap.MapApp with type key = accNumber

  type state = { mutable bal: t amount }
    invariant  { forall a :accNumber. mem a bal -> find a bal
>= 0 }
    by { bal = create() }

  val accounts :state
```

Declaring the state using a record type allows us to include the state invariant directly in the type definition, which makes it unnecessary to include it explicitly as a pre- and postcondition in function definitions (verification conditions will be created automatically for all state-changing functions, ensuring the preservation of the type invariant).

Note that:

- The `by` clause is mandatory. Its role is to provide a witness satisfying the type invariant. We simply provide the empty mapping as example (it is returned by the `create` function)
- Record fields are by default immutable (as in functional programming). Mutable fields must be explicitly identified as above

Both in the code and the spec, the balance value will have to be referred using record field notation. Note also the use of the `<-` assignment operator: since we are now using an applicative map type, the `add` function may not have side effects — in functional style, it takes a map and returns a new map).

```
  let open (n :accNumber) : ()
    requires { not mem n accounts.bal }
    ensures  { mem n accounts.bal /\ find n accounts.bal = 0 }
```

```
    ensures  { forall a :accNumber. mem a accounts.bal <-> mem
a (old accounts.bal) \/ a = n }
    writes   { accounts.bal }
  = accounts.bal <- add n 0 accounts.bal


  let deposit (n :accNumber) (x :amount) : ()
    requires { mem n accounts.bal /\ x > 0 }
    ensures  { find n accounts.bal = find n (old accounts.bal)
+ x }
    ensures  { forall a :accNumber. mem a accounts.bal  /\ a <
> n
                -> find a accounts.bal = find a (old accounts.
bal) }
    ensures  { forall a :accNumber. mem a accounts.bal <-> mem
a (old accounts.bal) }
    writes   { accounts.bal }
  = let baln = find n accounts.bal in
    accounts.bal <- add n (baln+x) accounts.bal
```

Exercises

1. Complete the definition of this alternative implementation, by writing all the remaining functions and their contracts (do not include the type invariant in the contracts). Observe carefully all verification conditions, in particular those pertaining to the type invariant.
2. Prove also the postcondition of the following main function

```
  let main ()
    ensures { find 3333 accounts.bal = 200 }
  = accounts.bal <- create()  ;
    open 1111 ;
    open 2222 ;
    open 3333 ;
```

```
    deposit 1111 100 ;
    deposit 2222 100 ;
    transfer 1111 3333 100 ;
    transfer 2222 3333 100
```

Challenge

Consider now that we want to prove properties involving the global assets held in the accounts:

- The `create` and `transfer` functions do not modify the total value of these assets;
- `deposit` and `withdraw` increase or decrease the total value by `x`

In order to state these properties we need a way to compute this global value (at logic level). The problem is that the domain of the map types is a set, and thus **not iterable**.

We may address this problem by including in the record type a *ghost field* of a list type (i.e. a field that is not meant for programming, only kept for specification/logic purposes), and including in the type invariant information tying the list to the domain of the map;

```
  use list.Mem


  type state = { mutable bal: t amount ; mutable ghost domain
: list accNumber }
    invariant  { (forall a :accNumber. mem a bal -> find a bal
>= 0) /\
                 (forall a :accNumber. MapApp.mem a bal <-> Me
m.mem a domain) }
    by { bal = create() ; domain = Nil }
```

```
val accounts :state
```

Now write a function to calculate the total value of the assets in the accounts (by traversing the domain list), and use it to extend the contracts with the properties expressed above.