B Assignment 4b

Ken Robinson

29th May 2006; Revised 1st June 2007

Name of assignment: ass4b
Assessment: 10 marks + 5 marks bonus.
Submission: `cs2111/bin/giveB ass4b VM
Note: the name of the development (VM) is unimportant

1 Purpose of this assignment

This assignment is concerned with the design refinement process. Refinement and implementation of the Vending machine developed in assignment 4a is used as an example, but the objective is not to simply produce the requested refinements. Rather, it is to layout the refinement in a way that makes it clear to the reader how you are going about your refinement. For that reason you are encouraged to put your development through several refinement steps, with each step dealing with some particular refinement, for example refinement of a single operation, or removal of parallel composition and use of sequential compositions. This assignment specification sets out some steps, but you should feel free to design the rest however is appropriate for you.

Proof obligations and proof Proof obligations are extremely useful during the refinement process, as undischargeable POs are identifying flaws in your design. These will range from typographical errors in your code to insecurities in your design. This particular exercise contains an implementation of the computation of maximal valid change. Some of the proof obligations for that are difficult, and proof of those POs is not expected. Away from that you should pay careful attention to the undischarged POs: either try to discharge them or at least understand what they are saying and determine whether they are proveable.

2 What you have to do

The ultimate objective of this assignment is to produce an implementation of Vending.mch through the following steps.

1. Revise Vending_ctx.mch.
2. Define the new machine CoinBox.mch.
3. Refine Vending.mch to VendingR.ref, including CoinBox.
4. Refine VendingR in one or more steps to a refinement that does not contain parallel composition or any other form of nondeterminism.

5. Refine to an implementation of Vending.

6. Refine to an implementation of Vending_ctx.

7. In the Translators environment translate all implementations to code.

8. In the Introduce menu create a new interface for an implemented machine, namely Vending. A template for the interface will be placed in your editor and you should provide value for machine parameters of Vending.

9. In the Generators environment select the itf button to create the interface.

10. In the Translators environment select the exe button to run the interface.

2.1 Revision of Vending_ctx

As in the previous assignment the context machine should contain:

The context contains:

SNACK an enumerated set containing an item called null or nosnack null that doesn’t represent any snack.

SNACKCODE New A function mapping snacks to a code. See section on enumerated sets for more information.

COIN an enumerated set of coin denominations. Make them OneDollar, TwoDollar, and FiveDollar, representing one, two and five dollar coins. You may choose money values and prices to be in either dollars or cents, as long as you are consistent across the whole development.

CoinValue a constant function that converts a coin denomination to its value expressed as a natural number (cents).

Revision: make CoinValue a total bijection.

<table>
<thead>
<tr>
<th>CONSTANTS</th>
<th>CoinValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROPERTIES</td>
<td>CoinValue ∈ COIN ↦ {1, 2, 5} ∧</td>
</tr>
<tr>
<td></td>
<td>CoinValue = {OneDollar ↦ 1, …}</td>
</tr>
</tbody>
</table>

nomoney Add a constant nomoney if you don’t already have one, with the following properties:

nomoney ∈ MONEY ∧ ∀(coin).(coin ∈ COIN ⇒ nomoney(coin) = 0)

MONEY a definition denoting a bag of COIN. Don’t use any bag machines, just use a total function. It’s probably easiest to use a definition in the DEFINITIONS section:

MONEY ≜ COIN → N;
MoneyValue a constant function that converts a bag of coins to their value in cents.

MoneyDiff a constant function that produces the difference between two bags of coins.

RESPONSE an enumerated set of appropriate responses.

The RESPONSE is optional as it will not be used in this assignment.

Additionally, add an operation coinValue:

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>value ← coinValue(coin) =</td>
</tr>
<tr>
<td>PRE coin : COIN THEN</td>
</tr>
<tr>
<td>value := CoinValue(coin)</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

2.2 CoinBox machine

It is sensible to put all operations involving the handling of coins and the coinbox in a separate machine. This could have been done from the beginning, but it is arguable that it works even better in association with the refinement of Vending.

The CoinBox machine should contain, from Vending:

- the variable coinbox;
- the operation AddCoinsF;
- the operation RemoveCoinsF;

and a new operation:

- change ← GiveChange(maxchange), an operation that contains the logic for giving optimal change, given the desired change maxchange. This operation contains the change giving specification found in the operation DispenseSnackF. This operation will be called from DispenseSnackF and CancelF, composed with the rest of the operation by sequential composition.

2.3 First refinement of Vending

Refine Vending by including CoinBox and promoting AddCoinsF and RemoveCoinsF. Replace all parallel composition by sequential composition.

2.4 Refinement of CoinBox

Refine CoinBox in one or more steps until all nondeterminism is removed.

To compute change you can use without proof the strategy of simply choosing the largest number of each denomination available to get the nearest possible change to the correct change. The reason this works is connected with some number theoretic properties of 5, 2, 1. Notice that this
MACHINE CoinBox (maxcoins)
CONSTRAINTS maxcoins ∈ N
SEES Vending_ctx
VARIABLES coinbox
INVARIANT
  coinbox ∈ MONEY ∧
  CoinCount(coinbox) ≤ maxcoins
INITIALISATION coinbox := nomoney

OPERATIONS
  AddCoinsF(coin, quantity) ≜
    pre coin ∈ COIN ∧ quantity ∈ N ∧
    CoinCount(coinbox) + quantity ≤ maxcoins then
    coinbox(coin) := coinbox(coin) + quantity
  end;
  RemoveCoinsF(coin, quantity) ≜
    pre coin ∈ COIN ∧ coin ∈ dom(coinbox) ∧
    quantity ∈ N ∧ quantity ≤ coinbox(coin) then
    coinbox(coin) := coinbox(coin) − quantity
  end;
  change ←− GiveChange(maxchange) ≜
    pre maxchange ∈ N then
    any chng where
      chng ∈ MONEY ∧ submoney(coinbox, chng) ∧
      MoneyValue(chng) = . . .
    then change := MoneyValue(chng) ||
    coinbox := MoneyDiff(coinbox, chng)
  end
  end
END

Figure 1: CoinBox machine
property doesn’t hold for any set of denominations. Suppose that in addition to 5, 2 and 1 there was also a 3. Suppose the correct change is 9. You would choose 1x5, 1x3 and 1x1, using the above strategy. But suppose that you didn’t have any 1 coins? Your simple strategy would yield 1x5 plus 1x3 and your change is 1 short. But suppose that you did have some 2 coins; you could have given exact change: 1x5 plus 2x2, but the simple strategy would block that. That sort of problem can’t happen with 5,2,1.

2.5 Implementation

Now refine the refinements of Vending and CoinBox to implementations using rename_Nfnc and rename_Nvar machines from the system library. You will also need to implement Vending_ctx:

2.6 A note on enumerated sets

Enumerated sets, like COIN and SNACK are convenient as they create named constants, but they have problems when it comes to implementation. The function machines in the system library only allow domains that are a subset of \( \mathbb{N} \), so you have a problem implementing a function like \( \text{coinbox} \in \text{COIN} \implies \mathbb{N} \), as enumerated sets are not a subset of \( \mathbb{N} \). So how do we implement \( \text{coinbox} \) in CoinBox? Here’s how:

```
IMPORTS coinbox_Nfnc(maxcoins, 5)
INVARIANT (CoinValue \circ \text{coinbox}_Nfnc) = \text{coinbox} 
  ... 
INITIALISATION
  \text{coinbox}_\text{STO}_\text{NFNC}(100, 0);
  \text{coinbox}_\text{STO}_\text{NFNC}(200, 0);
  \text{coinbox}_\text{STO}_\text{NFNC}(500, 0)
OPERATIONS
  \text{AddCoinsF}(\text{coin}, \text{quantity}) \equiv
  \text{VAR} \text{val IN}
  \text{val} \leftarrow \text{coinValue(coin)};
  \text{coinbox}_\text{ADD}_\text{NFNC}(\text{val}, \text{quantity})
END;
```

You will see that we implement the \( \text{coinbox} \) function as a function between coin values and coin quantity. That is quite adequate. But the \text{AddCoinsF} operation, as for all other operations that handle coins takes in the coin as an enumerated set value, which is nicer for the interface.

A similar thing will need to be done for functions having SNACK, or a subset, as the domain set. That’s why you will need the \text{SNACKCODE} function to map snacks into a code, like a barcode.

2.7 Advice

Keep things simple Use simple functions wherever possible. You might want to copy your current vending machine and simplify what you have. For example, you really only need a very simple function for each bag; you don’t need the BagOBJ machine, which is intended for
dynamic allocation of bags. And you might find that modelling a bag with a total function makes things easier.

You may have more than one bag in your specification of Vending: it's possible that you had two bags for coins and one for snacks. You definitely don't need a bag for snacks as there are no true bag operations involved in the vending machine: snacks are always processed as single snack, never multiple snacks. It is possible to motivate the use of two bags for coins, but that's your decision. In any case the bag of coins should be represented by a function, and probably a total function. For coins it is recommended that you have some bag functions such as MoneyDiff to remove one bag of coins from another bag of coins; MoneyValue to sum the total value of coins in a bag of coins; SubMoney a relation that determines whether one bag of coins is a sub-bag of another bag of coins; CoinCount to sum the value of all coins in a bag.