CSci 556: Multiparadigm Programming
Carrie's Candy Bowl Semantics

H. Conrad Cunningham

27 February 2017

Contents

Acknowledgements: I wrote the first version of these notes in Fall 2013 as comments in Lua code for the Carrie's Candy Bowl case study. This code was for an extended solution to a problem given on a take-home examination. In Fall 2014, I revised the comments and used them as the basis for separate documents.

In Spring 2017, I revised the documents and reformatted them to use Markdown, restated the semantics to use the "better semantics" documented in the code, and partially updated the document for use in a multi-language context.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and translate them to HTML and PDF.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of February 2017 is a recent version of Firefox from Mozilla.

Carrie's Candy Bowl ADT

Introduction

Carrie, the Department's Administrative Assistant, has a candy bowl on her desk. Often she fills this bowl with pieces of candy, which are quickly consumed by students and professors.

This case study describes the candy bowl as an abstract data type (ADT) suitable for implementation as Lua modules.

Notes:

1. This problem description differs slightly from the descriptions of the problem used in later Scala- and Haskell-based classes.
2. This problem description allows the implementations to use mutable state or to use only immutable state. Thus the specification may be imprecise about the possible effects of mutator operations.

Specifying Semantics

Operations

We identify four basic kinds of operations on abstract data types:

1. A *constructor* (also called a creator, factory, or producer) operation creates a new instance of the ADT.
2. A *mutator* (also called a setter, modifier, or command) operation returns the input instance with its state changed.
3. An *accessor* (also called a getter, observer, or query) operation accesses input instance's state and returns a value that characterizes the instance.
4. A *destructor* operator destroys an instance and "releases" its resources.

An ADT can be implemented in various ways. In most circumstances, it is best to keep an accessor as a pure function. That is, an accessor should not modify the state of the input instance in any observable manner. However, a mutator may be a pure function–returning a modified copy of the input instance–or it may be impure–directly modifying the value of the input instance.

In the imperative, object-oriented language Java, an abstract data type is typically defined using an `abstract class` or `interface` and implemented using a concrete `class` that extends the abstract class or implements the interface. The various ADT operations are the public methods of the class. An instance of the ADT is an instance of the concrete class whose instance variables hold the instance's state.

In the purely functional language Haskell, an abstract data type is typically implemented using a `module` with definitions of its types and functions. The module explicitly exports the public types and operations and hides other aspects of the specific implementation. Constructor functions create and return new instances of the ADT. These instances must be explicitly passed into the accessor, mutator, and destructor operations. A mutator operation returns a modified copy of the input instance. Currently, the module system does not support multiple implementations of the same interface; thus the overall abstract data type is represented by a set of modules all having the same set of exported operations.

Lua is a minimalistic language, but it is one that has powerful, flexible features. It is dynamically typed and mostly imperative. It offers a number of ways to implement abstract data types. As a result, it is challenging to specify ADTs formally.

This case study assumes we are implementing the ADT as a Lua module that exports functions for the public operations. However, the case study seeks to enable a number of different implementation techniques.

Contracts

The module defines the semantics of the Candy Bowl abstract data type (ADT) using

- *invariant* assertions to characterize valid states of its instances (between ADT operation calls)
- *precondition* and *postcondition* assertions to characterize the behavior of ADT operations

These assertions (or predicates or Boolean functions) are logical statements that must hold (i.e., be true) for an ADT instance to be valid.

If the precondition of an operation holds, then the operation must terminate with its postcondition true. The precondition characterizes valid values of the input arguments and other aspects of the program's state at the time of the call. The postcondition characterizes the value of the return value and any other effects caused by the operation. The postcondition often states the output values in terms of the input state.

The invariants must hold for an instance after execution of its constructor, before and after execution of mutator and accessor operations, and before execution of any destructor operation. Invariants are implicitly conjuncted (i.e., ANDed) to the preconditions and postconditions of the ADT's operations.

An ADT *interface invariant* specifies the valid value (i.e., state) of an ADT instance in terms of an abstract model of the ADT (and perhaps of primitive

operation in the module's interface). It must be the same for any implementation of the ADT. It does not reference the state of any implementation variables.

An ADT *implementation invariant* complements the interface invariant by defining the abstract model's state in terms of the state of an implementation's concrete variables and data structures.

We consider the invariants and the preconditions and the postconditions of all public operations of the ADT operations to form the *contract* between the users and developers of the ADT.

Concepts and notation

We use the following mathematical and logical concepts and notation to express the semantics of the ADT and its operations. Here, we use notation that can be typed into comments for computer programs, not the notation that would be typeset in a mathematics or logic textbook. The logic notation follows that popularized by Edsger Dykstra, David Gries, and others. (We do not use all of these concepts and notation in specification of the Candy Bowl abstract data type.)

The following gives an informal explanation of the concepts and notation.

- `(ForAll x : D(x) :: R(x))` denotes universal quantification. It is true if and only if assertion `R(x)` is true for *all* values `x` that satisfy assertion `D(x)`. If `D(x)` is omitted, then it does not constrain `x`.

  We can extend this notation to quantify over tuples of values such as in `(ForAll x,y,z : D(x,x,z) :: R(x,y,z))`.

- `(Exists x : D(x) :: R(x))` denotes existential quantification. It is true if and only if assertion `R(x)` is true for *at least one* value `x` that satisfies assertion `D(x)`. If `D(x)` is omitted, then it does not constrain `x`.

- `(# x : D(x) :: R(x))` denotes a count of all values `x` that satisfies assertion `D(x)` and assertion `R(x)`.

- `<=>` denotes logical equivalence. `p <=> q` is true if and only if the logical (Boolean) values `p` and `q` are equal (i.e., both true or both false).

- A *set* is an unordered collection of elements without duplicates. We use typical notation and operations on sets.

- A *bag* (also called a *multiset*) is an unordered collection of elements in which each value may occur one or more times. Here we use the following notation.

  - `{| |}` denotes the empty bag.

  - `{| 2, 3, 2, 1 |}` denotes a bag with four elements including two 2's, one 3, and one 1. The order is not significant! There may be one or more occurrences of an element.
  - A set can be considered a bag with at most one occurrence of an element.
  - We can formalize a bag as a function from the set of elements to the natural numbers (i.e., nonnegative integers).
- A *sequence* is an ordered collection of elements in which each value may occur one or more times.
- A *type* consists of a set of values and a set of operations. In some circumstances, we consider the type as a set of values.
- Membership. For a collection (i.e., a set, bag, sequence, or type) `C`, `x IN C` if only if element `x` occurs in collection `C`.
- `OCCURENCES(x,C)` denotes the number of times element `x` appears in collection `C`. For a set, this will be integer 0 or 1. For a bag or sequence, there many be any nonnegative number of occurrences.
- `CARDINALITY(C)` denotes the total number of occurrences of all elements in a collection `C`.
- `REPEAT(e,n)` denotes a bag containing exactly `n` occurrences of `e` or an empty bag if `n < 1`.
- Union.
  - For sets `C` and `D`, an element occurs in the set `C UNION D` if and only if it occurs in `C` or in `D` (or both).
  - Similarly, for bags `C` and `D`, an element has exactly `n` occurrences in bag `C UNION D` if and only if `n` is the *maximum* of the number of its occurrences in `C` and in `D`

    `{| 1, 1, 2, 1 }} UNION {| 2, 1, 1 |}` yields `{| 1, 1, 1, 2 |}`.
- Sum.
  - For bags `C` and `D`, an element has exactly `n` occurrences in bag `C + D` if and only if `n` is the *sum* of the number of occurrences in `C` and in `D`.

    `{| 1, 1, 2, 1 }} + {| 2, 1, 1 |}` yields `{| 1, 1, 1, 1, 1, 2, 2 |}`.
- Intersection.
  - For sets `C` and `D`, an element occurs in set `C INTERSECT D` if and only if it occurs in both `C` and `D`.

  - Similarly, for bags `C` and `D`, an element has exactly `n` occurrences in bag `C INTERSECT D` if and only if `n` is the *minimum* of the number of occurrences in `C` and in `D`.

    `{| 1, 1, 2, 1 }} INTERSECT {| 2, 1, 1 |}` yields `{| 1, 1, 2 |}`.

- Difference.
  - For sets `C` and `D`, an element occurs in set `C - D` if and only if it occurs in `C` but not in `D`.
  - Similarly, for bags `C` and `D`, an element has exactly `n` occurrences in bag `C - D` if and only if the elements occurs exactly `n` more times in `C` than in `D`,

    `{| 1, 1, 2, 1 }} - {| 2,1,1 |}` yields `{| 1 |}`.

- Subset.
  - For sets `C` and `D`, `C SUBSET_OF D` denotes that all the elements of `C` also occur in `D`.
  - For bags `C` and `D`, `C SUBSET_OF D` denotes that if any element has `n` occurrences in `C` and `m` occurrences in `D` then `n <= m`.
- TODO: Define Cartesian products, tuples, relations, functions?
- A tuple such as `(x,*)` appearing in a collection such as `{ (x,*) }` denotes element `x` grouped with all possible values of the second component. Note: We could also write `{ (x,*) }` using quantification as `{ (x,c) :: c IN some_domain }`.
- A *function* is a special case of a *relation* and relation is a set of ordered pairs (tuples). We sometimes manipulate functions or relations using set notation for convenience.
- A *total function* is defined for all elements of its domain. A *partial function* is defined for a subset of the elements of its domain.

Candy Bowl Specification

Abstract model

We represent a Candy Bowl instance as a mathematical bag of `CandyType` elements. If `bowl` is some representation of the Candy Bowl in the program, then the notation `Bag(bowl)` denotes the corresponding abstract model.

Interface invariant

Any valid instance of the Candy Bowl must satisfy the bag properties.

All Candy Bowl instances passed explicitly or implicitly to an ADT operation or returned explicitly or implicitly must be valid.

Constructive semantics

The preconditions and postconditions of each ADT operation are given below. These are part of the (implementation-independent) specification of the ADT.

We parameterize the specification with a domain set `CandyType`. We assume that assertion `validCandyType(c)` holds if and only if `c` represents a candy type supported by the module.

For convenience, we assume assertion `validCandyBowl(b)` holds if and only if `b` represents a bowl that satisfies the invariants. (Given our assumption that invariants are implicitly conjuncted with both preconditions and postconditions, use of this should not be necessary in most circumstances. However, we use it below to make the intention clear.)

In a postcondition:

- `Arg_XXX` denotes the value of explicit or implicit argument variable `XXX` at the time of the call of the operation.
- `Return` denotes the value returned explicitly by a operation's function.
- `Unchanged(Arg_b)` denotes that the value of variable `b` is not changed by the operation's execution.

Now, let's examine the operations.

1. Constructor function `CandyBowl(bowl)` returns a new candy bowl. If the argument `bowl` is (Lua value) `nil` or unspecified, then the new bowl is empty; otherwise, the new bowl has the same elements as the `bowl` argument.

   Note: The optional argument `bowl` was not in the original Fall 2013 problem description; I added it to illustrate use of optional arguments. If an argument is omitted, then the function sees it has having a `nil` value. For other languages, the one-argument function might need to be separate from the zero-argument function.

   - Precondition:

     ```
     bowl == nil OR validCandyBowl(bowl)
     ```

   - Postcondition:

     ```
     (if Arg_bowl == nil
         then CARDINALITY(Bag(Return)) == 0
     ```

```
    else Bag(Return) == Bag(Arg_bowl))
&& Unchanged(Bag(Arg_bowl))
```

Discussion: The one-argument operation is intended to be a "copy" constructor. (a) The new bowl should be a copy of the input bowl, not just another reference to the input (i.e., an alias). (b) The input `bowl` should not be changed.

These are tricky to specify formally because of Lua's dynamic typing, its passing of its `table` data structure by reference, and the relative openness of its `table` data structure to modification.

The `Unchanged(Bag(Arg_bowl))` conjunct on the postcondition addresses second intention.

For now, this specification does not address the copying/aliasing issue. Even if we require `Arg_bowl` and `Return` to denote different addresses, there is no good way to state that they do not share components.

2. Accessor function `has(bowl,candyType)` returns true if and only if `bowl` contains one or more pieces of type `candyType` candy. The argument `bowl` is not modified.

   - Precondition:

     ```
     validCandyBowl(bowl) && validCandyType(candyType)
     ```

   - Postcondition:

     ```
     Return == OCCURENCES(candyType,Arg_bowl) > 0
     && Unchanged(Arg_bowl)
     ```

3. Mutator function `put(bowl,candyType,n)` adds integer `n` pieces of `candyType` candy to the argument `bowl`. `n` defaults to 1 if `nil` or unspecified.

   Note: The optional argument `n` was not in the original Fall 2013 problem description. For other languages, the three-argument version might need to be a separate function from the two-argument function.

   - Precondition:

     ```
     validCandyType(candyType) && (n == nil or n > 0)
     ```

   - Postcondition:

     ```
     if n == nil
     then Bag(Return) == Bag(Arg_bowl) + {| candyType |}
     else Bag(Return) == Bag(Arg_bowl) + REPEAT(c,n)
     ```

   Discussion: The original problem specification allows this function to modify its `bowl` argument. We can add conjunct

   ```
   Unchanged(Arg_bowl)
   ```

to the postcondition to require a function without side effects. (Function `put` in the implementation modifies its argument; alternative function `putcp` returns a modified "shallow" copy.)

4. Mutator function `take(bowl,candyType)` removes one piece of `candyType` candy from the argument `bowl`.

   - Precondition:

     ```
     validCandyBowl(bowl_ && validCandyType(candyType) &&
     OCCURRENCES(candyType,Bag(Arg_bowl)) > 0
     ```

   - Postcondition:

     ```
     Bag(Return) == Bag(Arg_bowl) - {| candyType |}
     ```

   Discussion: The original problem specification allows this function to modify its `bowl` argument. We can add conjunct

   ```
   Unchanged(Arg_bowl)
   ```

   to the postcondition to require a function without side effects. (Function `take` in the implementation modifies its argument; alternative function `takecp` returns a modified "shallow" copy.)

5. Accessor function `howMany(bowl,candyType)` returns the number of pieces of candy that are in the `bowl` overall if `candyType` is nil; otherwise, the function returns the count of the pieces of `candyType` candy that are in the bowl.

   Note: Argument `candyType` was not optional in the original Fall 2013 problem description. For other languages, the one- and two-argument functions would need to be separate.

   - Precondition:

     ```
     validCandyBowl(bowl) &&
     (candyType == nil OR validCandyType(candyType))
     ```

   - Postcondition:

     ```
     ( if candyType == nil
       then Return == CARDINALITY(Bag(Arg_bowl))
       else Return == OCCURRENCES(candyType,Bag(Arg_bowl) )
     && Unchanged(Arg_bowl)
     ```

6. Accessor function `isEmpty(bowl)` returns true if and only if the `bowl` is empty.

   - Precondition:

     ```
     validCandyBowl(bowl)
     ```

   - Postcondition:

```
Return == CARDINALITY(Bag(Arg_bowl)) == 0
&& Unchanged(Arg_bowl)
```

7. Accessor function `inventory(bowl)` returns a list-style table of pairs `{candyType, count}`.

   Note: This function describes a specific Lua data structure to be returned, so the specification uses Lua notation.

   - Precondition:

     ```
     validCandyBowl(bowl)
     ```

   - Postcondition:

     ```
     validInventory(Return) &&
     (ForAll c,n: OCCURRENCES(c,Bag(Arg_bowl)) == n && n > 0 ::
         (Exists i: i > 0 :: Return[i] == {c,n}) )) &&
     (ForAll i,c,n: 1 <= i <= #Return && Return[i] == {c,n} ::
         OCCURRENCES(c,Bag(Arg_bowl))== n)) &&
     Unchanged(Arg_bowl)
     ```

     where

     ```
     validInventory(inv) ==
         inv ~= nil && type(inv) = "table" &&
         (ForAll i: i < 1 or i > #inv :: inv[i] == nil) &&
         (ForAll i: 1 <= i <= #inv ::
             (Exists c,n: inv[i] == {c,n}::
                           validCandyType(c) && n > 0) )
         &&
         (ForAll i: 2 <= i <= #inv :: inv[i-1][1] < inv[i][1]) )
     ```

   For example, if `candyType` is denoted by a string and there are two Snickers and one Hershey Kiss in the bowl, then the list returned would be something like `{ {'Hershey Kiss', 1}, {'Snickers', 2} }`.

8. Constructor function `toBowl(inv)` takes an inventory `inv` (as returned by the `inventory` operation and returns the corresponding bowl.

   Note: This operation was not in the original problem description. It taskes a specific Lua data structure as input, so the specification uses Lua notation.

   - Precondition:

     ```
     validInventory(inv)
     ```

   - Postcondition:

     ```
     (ForAll c,n: OCCURRENCES(c,Bag(Return)) == n && n > 0 ::
         (Exists i: 1 <= i && i <= #inv :: inv[i] == {c,n}) ))  &&
     (ForAll i,c,n: 1 <= i <= #inv && inv[i] == {c,n} ::
     ```

```
        OCCURRENCES(c,Bag(Return)) == n)) &&
    Unchanged(inv)
```

- Property:

```
(ForAll bowl :: validCandyBowl(bowl) ::
    Bag(toBowl(inventory(bowl))) == Bag(bowl) )
```

9. Mutator function `combine(bowl1,bowl2)` returns the bowl resulting from pouring `bowl1` and `bowl2` together to form a new bowl.

   - Precondition:

     `validCandyBowl(bowl1) && validCandyBowl(bowl2)`

   - Postcondition:

     `Bag(Return) == Bag(Arg_bowl1) + Bag(Arg_bowl2)`

   Discussion: The original problem specification allows this function to modify the `bowl1` and `bowl2` arguments. We can add conjuncts

   `Unchanged(Arg_bowl1) && Unchanged(Arg_bowl2)`

   to the postcondition to require a function without side effects. (Function `combine` in the implementation modifies its first arguments; alternative function `combine2` returns a modified "shallow" copy.)

For other languages, the aspects of the contracts related to the abstract bag model should be the same. Of course, references to specific Lua features such its `table` data structure, value `nil`, and optional arguments would need to be changed appropriately.

Data Representations

Lua hashed Version

The first version of the module represents the Candy Bowl ADT with a Lua hash table that maps `candyType` key values to the count of the pieces of that type. In addition, the special key `SIZE` maps to the count of the total number of pieces in the table.

An ADT *implementation invariant* for Candy Bowl `bowl` can be stated as

```
(ForAll c: validCandyType(c) ::
    if bowl[c] ~= nil
      then bowl[c] == OCCURRENCES(c,Bag(bowl))
      else OCCURRENCES(c,Bag(bowl)) == 0)
&& bowl[SIZE] == CARDINALITY(Bag(bowl))
```

where `validCandyType(c) == (c ~= nil && c ~= SIZE)`.

Lua list Version

The second version of the Candy Bowl ADT module has the same interface as the hashed version, but it uses a different internal data representation. This version uses with an unsorted, list-style table of `candyTypes`.

An ADT *implementation invariant* for Candy Bowl `bowl` can be stated as

```
(ForAll i : 1 <= i <= #bowl :: validCandyType(bowl[i]))  &&
(ForAll c : validCandyType(c) ::
  howMany(bowl,c) == (Sum i : 1 <= i < #bowl && bowl[i] == c :: 1))
```

where `validCandyType(c) == (c ~= nil)`.