Specifying and Runtime Monitoring Java Classes with CONGU 2.0

V. T. Vasconcelos, A. Lopes and I. Nunes

Lecture Notes
Algoritmos e Estruturas de Dados

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

March 2015

Contents

List of Figures

Chapter 1

Introduction

These lecture notes describe an approach to the specification of Abstract Data Types and the runtime monitoring of their Java implementations aiming at determine whether these implementations behave as required. This approach is supported by the version 2.0 of the CONGU tool.

An *Abstract Data Type* (ADT) describes a set of abstract values and is defined by a set of operations and by pre-conditions and constraints on the effects of those operations. An implementation of an ADT in Java requires:

- the definition of an Application Programming Interface (API) describing the signature of constructors and methods that implement the ADT operations and the corresponding contracts, including how the erroneous situations are signalled with exceptions;
- the definition of a concrete class implementing that API;
- optionally, the definition of an interface capturing the decisions about the API.

These notes start with a rather simple data structure (stack) and go on with other data structures featuring in any standard textbook (e.g., queues, binary trees, sorted sets and maps). The intuition about these abstract data types must be obtained from an alternative source, for example the course's textbook.

For all of the addressed abstract data types we present a specification module, which include a self-contained collection of specifications, and a possible refinement of the module into Java classes.

Various exercises are presented that either extend a specification or a refinement into Java classes. Students are encouraged to try solving them. The material in these notes should be complemented with that on the course's web page `http://moodle.ciencias.ulisboa.pt` and also the resources presented on `https://`

www.di.fc.ul.pt/~mal/aed/AED%20ADTs.html. The tool and its users guide, further examples and documentation can be found on http://gloss.di.fc.ul.pt/quest/tools.

Chapter 2

Stacks

This chapter shows how to build a specification for a stack of arbitrary elements, and how to relate this specification with a particular Java class.

2.1 Constructing and Observing Stacks

Figure 2.1 presents a specification for stacks of elements of an arbitrary type. This is because, as indicated by [Elem] in the first line, the specification is parameterised by the specification Elem and this specification, presented in detail in Section 2.4, defines a data type without constraints.

Specifications define types of data, called **sorts**. Each specification defines exactly one sort, Stack[Elem] in this case.

After **sorts**, we specify the *signatures* for all the operations the specification defines. A signature defines the name, the arguments, and the result of the operation. For example

```
push: Stack[Elem] Elem —> Stack[Elem];
```

says that push is the name of an operation that requires two parameters (of sorts Stack[Elem] and Elem), and returns a value (of sort Stack[Elem]). The specification for sort Elem must be defined separately (see Section 2.4). The below signature introduces an operation that builds a Stack[Elem], requiring no input.

```
make: —> Stack[Elem];
```

Operations are declared in three separate sections: **constructors**, **observers**, and **others**. Constructors form a minimal set of operations needed to build any conceivable value of the sort. Sort Stack[Elem] uses two constructors: one to build a brand new Stack[Elem], the other to construct a new Stack[Elem] given another Stack[Elem] and an Elem.

```
specification Stack[Elem]
  sorts
    Stack[Elem]
  constructors
    make: --> Stack[Elem];
    push: Stack[Elem] Elem --> Stack[Elem];
  observers
    top: Stack[Elem] -->? Elem;
    pop: Stack[Elem] -->? Stack[Elem];
    isEmpty: Stack[Elem];
  domains
    S: Stack[Elem];
    top(S) if not isEmpty(S);
    pop(S) if not isEmpty(S);
  axioms
    S: Stack[Elem];  E: Elem;
    top(push(S, E)) = E;
    pop(push(S, E)) = S;
    isEmpty(make());
    not isEmpty(push(S, E));
end specification
```

Figure 2.1: A specification for a stack of arbitrary elements.

Observers are the operations used to analyze (to dissect, to disassemble, to deconstruct) a given value. Observers must have at least one parameter. Also, the first parameter must be of the sort under specification, Stack[Elem] in our example. For the stack specification, we have chosen two observers: top that returns the element at the top of a Stack[Elem], and pop that returns the Stack[Elem] obtained by removing the element at the top of a stack.

There is a special class of operations that are used, not to build elements of a given sort, but to evaluate a condition. We call them *predicates*. Given that predicates return nothing, we omit the arrow altogether. In our case we have a single predicate

```
isEmpty: Stack[Elem];
```

used to inquire whether a given stack contains no element. Such a predicate requires a Stack[Elem] and, intuitively says "yes, the stack contains no element" or "no, the stack contains at least one element".

Not all operations are *total*: some operations *may not* be defined for all possi-

ble values of the parameters. In the case of stacks, we do not know how to pick the element at the top of an empty stack. Similarly, we cannot remove the element at the top of an empty stack. We must then declare that these operations may not be available for any conceivable stack. We do this in two parts. First, in the signature of the operation, we use a *partial function arrow* $--$>?, rather than a *total function arrow* $--$>, as in:

```
top: Stack[Elem]-->? Elem;
```

Then, for each partial operation, we add one or more entries in the **domains** section. Such a section sets the domain for the partial operations declared in the various signature sections.[1] It starts by declaring the variables needed to describe the domains. In our case we need a variable S of sort Stack[Elem], written:

```
S: Stack[Elem];
```

Variable S represents, within the **domains** section only, any conceivable stack: S: Stack[Elem] can be read "for all stack S". We then say that both the top and the pop operations must be *defined at least* for non empty stacks, by using the domain conditions below.

```
top(S) if not isEmpty(S);
pop(S) if not isEmpty(S);
```

Since top operation (and similarly for pop) is not required to be defined over an empty stack, we do not have any axiom of the form top(empty()=.... This means that the result of top over an empty stack is not specified. We leave to the programmer that implements our specification to decide what to do in this case.

The various operations of a sort are not unrelated. For example, if we push an element E onto a stack and then ask for the element at the top of the stack, we expect to obtain the exact element E. By relating the various operations, the axioms provide a meaning, or a semantics, for the operations. The dependencies between the different operations are described in the **axioms** section. Such a section starts by declaring the variables needed for the axioms, as in the case of the **domains** described above. In our case we need a variable S of sort Stack[Elem], and a variable E of sort Elem. Notice that there are two distinct scopes in a specification: one for **domains** and one for **axioms**. A variable declared in one scope is not visible in the other; variable S in Figure 2.1 is one such example.

In simple cases we require *one axiom per observer, per constructor*. Nonetheless, an axiom for top(make()) makes no sense, since top may not be defined on empty stacks (more precisely, because an empty stack is not part of the minimal domain of the top operation). Hence only one axiom makes sense for observer top, namely, that of top(push (...)) . Axiom

[1] Concrete implementations may then relax the domain, but not restrict further.

```
top(push(S, E)) = E;
```

says that the element at the top of a stack where we have just pushed an element E, is precisely E. An operational reading is as follows: if one pushes an element E onto a stack, and then reads the element at the top, then one obtains E. We leave to the reader the interpretation of the only axiom for pop.

```
pop(push(S, E)) = S;
```

The next two axioms relate the observer isEmpty with the two constructors make and push.

```
isEmpty(make());
not isEmpty(push(S, E));
```

The first axiom says that a newly made Stack[Elem] is empty, whereas the second says that it is false that the result of pushing any Elem on a given Stack[Elem] yields an empty stack. In other words, pushing something into a stack yields a non-empty stack.

2.2 Adding a Size Operation

Our next example improves the Stack[Elem] specification in Figure 2.1 by providing an operation to obtain the number of elements in a stack; the result is in Figure 2.2. The signature is easy: size requires a Stack[Elem] and returns an integer, for which we have chosen the primitive sort **int**:

```
size : Stack[Elem] ---> int;
```

Sort **int** is primitive. It comprises two equality operators (=, !=), four relational operators (<, >, <=, >=), two additive operators (+, −), three multiplicative operators (∗, /, %), and a unary operator (−), in this order of precedence. The **min**(_, _) and **max**(_, _) operators are also available.

Under which category do we classify the new operation? It is certainly not a constructor, for it does not return a Stack[Elem]. Since it allows to observe a property of stacks (the number of elements), we classify it under the **observers** heading.

The next step is defining the minimal domain for the operation. Can we ask for the size of no matter what Stack[Elem]? Certainly yes; hence no entry under this section (and hence the choice of the total function arrow −−> in the signatures section).

Since size is defined for arbitrary stacks, we need as many axioms as there are constructors; two in our case, one for size(make()) and another for

```
specification Stack[Elem]
  sorts
    Stack[Elem]
  constructors
     make: --> Stack[Elem];
     push: Stack[Elem] Elem --> Stack[Elem];
  observers
     top: Stack[Elem] -->? Elem;
     pop: Stack[Elem] -->? Stack[Elem];
     size: Stack[Elem] --> int;
  others
     isEmpty: Stack[Elem];
  domains
     S: Stack[Elem];
     top(S) if not isEmpty(S);
     pop(S) if not isEmpty(S);
  axioms
     S: Stack[Elem];  E: Elem;
     top(push(S, E)) = E;
     pop(push(S, E)) = S;
     size(make()) = 0;
     size(push(S, E)) = 1 + size(S);
     isEmpty(S) iff size(S) = 0;
end specification
```

Figure 2.2: A specification for a stack with a size operation.

size(push (...)) . The size of a just made stack is zero; that of a stack S where we have just pushed an element is one plus the size of S. We write all this as:

```
size(make()) = 0;
size(push S, E)) = 1 + size(S);
```

Our last step is to check whether the new observer has made another observer redundant. Can we define top in terms of size? and what about pop? Certainly not; but isEmpty has a close relationship with size. In fact an empty stack is a stack with zero elements.

The third category of signatures, **others**, is used in this case. Rather than discarding the isEmpty operator, we re-classify it as a **others** operation, and replace the two axioms for isEmpty in Figure 2.1 with a single axiom:

```
isEmpty(S) iff size(S) = 0;
```

```
specification Elem
  sorts
    Elem
end specification
```

Figure 2.3: A specification for the elements in a stack.

Notice that term equality uses =, whereas for predicates we must use **iff**. More precisely, the above axiom is short for the two axioms below.

```
isEmpty(S) if size(S) = 0;
not isEmpty(S) if size(S) != 0;
```

2.3 Observer or Others?

In Figure 2.2 we classify the size operation observer and isEmpty as **others**. Moving isEmpty from **observers** to **others** allows for a much more concise axiom.

An alternative specification classifies both size and isEmpty as **observers**. The choice affects only the form of the axioms. The properties of observers must be expressed against the constructors, as in:

```
isEmpty(make());
not isEmpty(push(S, E));
```

those of others can *also* be expressed against an arbitrary element of the sort, as in,

```
isEmpty(S) iff size(S) = 0;
```

In general, we suggest the choice embodied by Figure 2.2: define size as **observers** and isEmpty as **others**, for the reason that isEmpty is indeed a *derived* operation: if, for some reason the isEmpty operation is not defined, we can always use size(S) = 0 instead of isEmpty(S).[2]

2.4 Specifying Elem

Specification Stack[Elem] is parameterised by the specification Elem. We must then write a specification for it.

[2]This choice affects only the strategy employed by ConGu to convert axioms into verification conditions.

Parameters allow us to impose constraints over the concrete types we can use when instantiate parameterised data types. What do we require from the elements of a stack? Nothing special; only that they exist. We are happy with a completely underspecified type, composed of the sort alone. The specification Elem is then the shortest possible specification: it declares a sort, and leaves all other sections empty (we do not even need to write the names of the sections). Figure 2.3 describes such a specification. Since this type of specification is quite useful, CONGU includes the built-in specification Element, which also only defines a sort.

2.5 Modules

The meaning of symbols external to a specification (sort Elem in the stack specification, for example) is only fixed when the specification is embedded, as a component, in a *module*. A module defines a surjective function from a set of names N into specifications, such that all symbols (sorts, operations and predicates) are provided by some specification in the module. For example, a module for the specifications in this chapter can be given by the mapping

$$\text{Stack.spc} \mapsto \text{Figure 2.2}$$
$$\text{Elem.spc} \mapsto \text{Figure 2.3}$$

Additionally, a module defines whether each specification is a core specification or a parameter specification. Core specifications define the data types that need to be implemented. Parameter specifications are those just used for imposing constraints on the parameters.

A file directory (implicitly) defines a module, where N is the set of .spc filenames in the directory, and the associated specifications are the contents of the files. All specifications are considered core with the exception of specification Element that is by default considered a parameter specification.

In this way, a directory including two files named Stack.spc and Elem.spc, containing the text in Figures 2.2 and 2.3, respectively, defines a module that is not correct because it defines that Elem is a core specification. The module we need for the rest of this section is described by the specification module presented below. In section **core** we indicate the names of the core specifications and parameter specifications are identified in section **parameter**.

```
module
   core
      Stack
   parameter
      Elem
end  module
```

Figure 2.4: A specification module for a stack.

2.6 From Specifications to Java Classes

Suppose we have developed a revolutionary implementation of a stack.

2.6.1 API

The API of class RevolutionaryStack (for the rest is patent pending) is presented in Figure 2.5. Since methods push and pop are **void**, we can easily conclude that this class is a mutable implementation of a stack. Also notice that the class defines pre-conditions for methods peek and pop. This means that clients of this class should not call these methods over empty stacks since no guarantees are given in this case (not even that the execution of the method terminates).

Class RevolutionaryStack also defines a pre-condition for method push requiring that **null** is not given as input. This means that stacks of type RevolutionaryStack do not take **null** as elements. Notice that this is not inconsistent with the fact that operation push was declared total in the specification Stack[Elem] since **null** is a value that does not correspond to any abstract value.

2.6.2 Refinement mapping

Now suppose that we would like to check whether our revolutionary implementation conforms to the specification in Figure 2.2. On the one hand, there are specifications Stack[Elem] and Elem (more precisely, the module defined in the previous section), and on the other hand, we have a Java generic class RevolutionaryStack with type parameter E. In order to check conformance between the two we need to formally specify how they relate.

Class RevolutionaryStack is expected to implement specification Stack[Elem] and the type parameter E corresponds to the sort Elem. As in Java generic classes, the type variables used in a refinement are declared in the beginning between angle

```
public class RevolutionaryStack<E> implements Cloneable {
  /** Create an empty stack. */
  public RevolutionaryStack () {...}
  /**
   *Insert an element at the top of the stack.
   *@requires e!=null;
   */
  public void push (E e) {...}
  /**
   *Remove the top element from the stack.
   *@requires !isEmpty();
   */
  public void pop () {...}
  /**
   *Inspect the element at the top of the stack.
   *@requires !isEmpty();
   */
  public E peek () {...}
  /**@return the number os elements in the stack. */
  public int size () {...}
  /**@return whether the stack is empty. *//
  public boolean isEmpty () {...}
  public boolean equals (Object other) {...}
  public RevolutionaryStack<E> clone () {...}
 ...
}
```

Figure 2.5: A Java generic class implementing a stack.

brackets.

Also, on the one hand we have data type operations (such as pop and top), and on the other hand we have class methods (such as pop and peek). How do these relate? Method pop is supposed to implement the operation with the same name, and method peek is intended as an implementation of operation top. All this is defined in the *refinement mapping* presented in Figure 2.6.

For each module we must define a refinement mapping. Refinement mappings map specifications into Java types and specification operations and predicates into constructors and methods in these types. Core specifications are mapped into types defined by classes and parameter specifications are mapped to type variables. That sort Elem is refined into type variable E, and that sort Stack[Elem] is

```
refinement<E>
  Elem is E

  Stack[Elem] is RevolutionaryStack<E> {
    make: --> Stack[Elem] is RevolutionaryStack();
    push: Stack[Elem] e:Elem --> Stack[Elem] is void push(E e);
    pop: Stack[Elem] -->? Stack[Elem] is void pop();
    top: Stack[Elem] -->? Elem is E peek();
    isEmpty: Stack[Elem] is boolean isEmpty();
    size: Stack[Elem] --> int is int size();
  }
end refinement
```

Figure 2.6: A refinement mapping for the Stack module

refined into generic class RevolutionaryStack, is written as follows.

```
Elem is E
Stack[Elem] is RevolutionaryStack<E>
```

After each of these declarations there is an optional section, enclosed in braces, describing the operation bindings. Sort Elem contains no operation, so we omit the section and its braces. Sort Stack[Elem] contains five operations and one predicate, so we expect to see six entries in the corresponding section. The very first links operation make with the class constructor. We write:

```
make: --> Stack[Elem] is RevolutionaryStack();
```

At the left of keyword **is** we describe the signature as in the specification in Figure 2.2; at the right, the Java method, as in a Java class, except that we omit the visibility modifier (which must be **public**, very much like the methods in Java interfaces). For Java methods that require parameters, we must also describe the appropriate correspondence between the parameters to the specification operation and those in the method. This we do by annotating the parameter sort in the specification with an identifier.[3] For example, the refinement for operation push in Figure 2.2:

```
push: Stack[Elem] Elem --> Stack[Elem];
```

becomes, when annotated,

[3]The reader might wonder why this annotation is necessary. Imagine an operation with two parameters of the same type, refined into a method where the order of the parameters is reversed: op: S x:E y:E --> S **is void** m(Object y, Object x);

```
push: Stack[Elem] e:Elem --> Stack[Elem];
```

which we include in the refinement in Figure 2.6 as:

```
push: Stack[Elem] e:Elem --> Stack[Elem] is void push(E e);
```

Operation push is refined into a method with the same name. But that need not be the case. We decided to map operation top into method peek, and predicate isEmpty into method isEmpty (because we want to keep in line with the Java API nomenclature). We obtain:

```
top: Stack[Elem] -->? Elem is E peek();
```

Finally, predicates can only be mapped into **boolean** methods. That is the case with isEmpty. We get:

```
isEmpty: Stack[Elem] is boolean isEmpty();
```

Exercise 1 *Modify the specification in Figure 2.2 so that the ADT* Stack *supports an operation* clear *that removes all elements from a given stack. Change the implementation of* Stack *and the refinement in Figure 2.6 accordingly.*

Chapter 3

Queues

This chapter describes a specification for a queue of arbitrary elements. The queues we are interested in provide the following operations.

make to build an empty queue;

enqueue to add an element to the back of queue;

isEmpty to check whether the queue is empty;

front to obtain the element at the front of the queue;

dequeue to discard the element at the front of the queue.

3.1 Building a Specification

Figure 3.1 presents a parameterised specification for a queue of elements of an arbitrary sort.

The specification defines a sort, which we decided to call Queue[Element]. We must then classify the five operations above in the three categories(**constructors**, **observers**, and **others**), for this choice constrains the form of the axioms that we shall write thereafter.

We start by writing their signatures, inspired by the description above. To build an empty queue no parameter is required; the result of the operation is a queue(which turns out to be empty); we have:

```
make: --> Queue[Element];
```

```
specification  Queue[Element]
  sorts
    Queue[Element]
  constructors
    make: --> Queue[Element];
    enqueue: Queue[Element] Element --> Queue[Element];
  observers
    front: Queue[Element] -->? Element;
    dequeue: Queue[Element] -->? Queue[Element];
    isEmpty: Queue[Element];
  domains
    Q: Queue[Element];
    front(Q) if not isEmpty(Q);
    dequeue(Q) if not isEmpty(Q);
  axioms
    Q: Queue[Element];  E: Element;
    front(enqueue(Q, E)) = E when isEmpty(Q)
                           else front(Q);
    dequeue(enqueue(Q, E)) = Q when isEmpty(Q)
                             else enqueue(dequeue(Q), E);
    isEmpty(make());
    not isEmpty(enqueue(Q, E));
end specification
```

Figure 3.1: A specification for a queue of arbitrary elements.

To enqueue we need a queue and an element to place in the queue; in return we get another queue which also contains the newly added element. We have to decide on a name for the sort of the elements in the queue. Let it be Element; we get:

```
enqueue: Queue[Element] Element --> Queue[Element];
```

Operation isEmpty is a predicate; it requires a queue, as in:

```
isEmpty: Queue[Element];
```

To obtain the element at the front of the queue we need a queue. In return, we get an element:

```
front: Queue[Element] --> Element;
```

Finally, to remove an element from a queue, we obviously need a queue; in return we obtain a queue, as in:

```
dequeue: Queue[Element] --> Queue[Element];
```

```
public class ArrayQueue<E> implements Cloneable {
  public ArrayQueue () {...}
  public void offer (E e) {...}
  public boolean empty () {...}
  public E element () {...}
  public void remove () {...}
  public boolean equals (Object other) {...}
  public void clone () {...}
  ...
}
```

Figure 3.2: A Java generic class implementing a queue.

What are our constructors? We rule out isEmpty and front, for they do not return a Queue[Element]. The next question we should ask ourselves is "Are all the three remaining operations absolutely necessary to construct *any* conceivable queue?" Consider the queue Q obtained from queue Q' via operation dequeue. Can we obtain Q differently? It seems clear that, the element that dequeue discarded must have been placed in the queue with enqueue. Hence the queue resulting from the dequeue operation can be obtained differently, namely by not enqueueing the element in the first place.

There remains to classify three operations(isEmpty, front, and dequeue) according to the two categories **observers** and **others**. Can we define any one of them in terms of the other two alone? It does not look like. We classify them all under the section **observers**, and conclude that there is no operation to classify under **others**.

Next step: the domains for the various operations. We must be particularly attentive to functions that are not defined at arbitrary values of parameter. A little thought leads us to realize that the only problematic cases are when we ask for the element at the front of an empty queue, and when we ask to dequeue that element. We use a partial function arrow $--{>}?$ in the signatures of the operations, and record the restrictions in the **domains** section:

```
front(Q) if not isEmpty(Q);
dequeue(Q) if not isEmpty(Q);
```

The final, and most important step, is to write the axioms. By now you should have learned the method. First observers versus constructors, that is, observing the result of applying a constructor. Given that we have two constructors, and three observers, we should have six axioms in total. Notice however that front and dequeue are not applicable to empty queues. That reduces the number of

axioms to four.

Is a queue obtained via make empty? and the queue obtained with an enqueue?

```
isEmpty(make());
not isEmpty(enqueue(Q, E));
```

Given that we cannot observe the front of a queue obtained with make, we concentrate on queues obtained via enqueue. So we have just added an element E to the *back* of queue Q, and now we are asking the element at the *front*. The easy case is when the queue has a single element: E is our element. To say that the queue, after insertion, has a single element is to say that it was empty before insertion, that is when isEmpty(Q). We write this as follows.

```
front(enqueue(Q, E)) = E if isEmpty(Q);
```

And when Q is not empty? Then we ask for the front of the queue "old" Q. Here is the axiom:

```
front(enqueue(Q, E)) = front(Q) if not isEmpty(Q);
```

The two axioms can be combined into one that describes the result of obtaining the element at the front *when* isEmpty(Q) and otherwise.

```
front(enqueue(Q, E)) = E when isEmpty(Q)
                       else front(Q);
```

Our last axiom relates dequeue to enqueue. We are looking for an axiom whose left hand side is dequeue(enqueue(Q, E)). Once again, the easy case is when Q is empty: adding an element and removing it afterward yields Q. The not so easy case is when queue Q is not empty. We have to dequeue Q. But remember that we want to dequeue enqueue(Q, E), and not Q. We must then put E back into the result of dequeue(Q). We obtain the following axiom.

```
dequeue(enqueue(Q, E)) = Q when isEmpty(Q)
                         else enqueue(dequeue(Q), E);
```

Specification Queue[Element] has the specification Element as parameter. As mentioned before, this is a built-in specification which only defines the sort Element and, hence, we are not imposing any constraint over the data types that can be used to instantiate Element in Queue[Element].

3.2 Refining the Queue Specification

As in the case of stacks, we must prepare a *module* to refine. A directory containing the file named Queue.spc defines the module we need for the rest of this section. Recall that specification Element is built-in and is considered a parameter

```
refinement <E>
  Element is E
  Queue[Element] is ArrayQueue<E> {
    make: --> Queue[Element] is ArrayQueue();
    enqueue: Queue[Element] e:Element --> Queue
                        is void offer(Object e);
    front: Queue[Element] -->? Element is E element();
    dequeue: Queue[Element] -->? Queue[Element] is void remove();
    isEmpty: Queue[Element] is boolean empty();
  }
end refinement
```

Figure 3.3: A refinement mapping for the Queue module.

specification by default. Part of the class we want to monitor is in Figure 3.2. The refinement is described in Figure 3.3, associating sort Element with type variable E, and sort Queue[Element] with generic class ArrayQueue. After reading Section 2.6, the refinement should be easy to interpret. We have made operation make correspond to a constructor in class ArrayQueue—, operation offer correspond to method enqueue, and so on.

Exercise 2 *Add a* size *operation to the* Queue *specification in Figure 3.1. How do you classify your operations now?*

Exercise 3 *Add a* moveToRear *operation to the* Queue *specification in Figure 3.1 that moves the element currently at the front of the queue to the rear of the queue.*

Exercise 4 *Add to the specification in Figure 3.1 an operation to remove all elements from a given queue (cf. Exercise 1 for stacks).*

Chapter 4

Binary Trees

There are only two kinds of binary trees: empty trees and compound trees. For the former there is nothing else to say, all empty binary trees are alike. A non-empty binary tree, on the other hand, is made of a datum, stored in the root, and two binary trees, the left and the right subtrees. This simple observation leads us to a data structure with two constructors: one for an empty tree, the other for a compound tree. What can we ask about such trees? What facets can we observe? Given an arbitrary tree, the very first question that comes to our mind is "is it empty, or is it compound?". If empty, there is nothing else to ask, for empty trees are unstructured. However, if the tree is compound, then we can also ask for the datum at the root, and for the left and the right subtrees. We have just identified the four observers we are going to work with.

Having identified the minimum set of operations on binary trees, we can think of multiple interesting operations on trees. For example, a leaf is tree with a single node, that is a non-empty tree without left or right subtrees. We can also think of the height of a tree, or the number of occurrences of an element in a tree, or even other properties of trees such as being balanced, full, or complete. For the purposes of this chapter let us fix the following operations, noted below together with their informal description.

empty to build an empty tree;

make to build a tree given a datum, and two trees;

isEmpty to check whether the tree is empty;

root to obtain the root of a non-empty tree;

leftSubtree to obtain the left subtree of a non-empty tree;

rightSubtree to obtain the right subtree of a non-empty tree;

isLeaf does this tree contain a single node?

height to obtain the number of nodes on the longest path from the root to a leaf;

isBalanced a tree is balanced when the height of its subtrees differ at most by one, and the subtrees are themselves balanced.

4.1 Building a Specification

Figure 4.1 presents a parameterised specification for a binary tree of arbitrary elements. The classification of the various operations according to the three available categories(**constructors**, **observers**, and **others**) is discussed above: constructors are empty and make; observers are isEmpty, root, leftSubtree and rightSubtree. The remaining operators are all classified as others, allowing for extra flexibility in their axioms.

For the **domains** section, we look for operations which may not be defined for arbitrary values of the parameters. Since all parameters are of sort BinaryTree[Element], we look for operations which may not be defined for empty trees. Clearly the only problematic operations are those that observe non-empty trees, for they cannot be applied to this kind of trees. We write that root, leftSubtree and rightSubtree are defined for non-empty trees as follows.

```
root(T) if not isEmpty(T);
leftSubtree(T) if not isEmpty(T);
rightSubtree(T) if not isEmpty(T);
```

One must not forget to annotate the signatures of these operations with a partial arrow, as for example:

```
root: BinaryTree[Element] -->? Element;
```

The final step is axiom writing. As usual we start with the "observing the constructors" axioms. With two constructors and four observers one aims at eight axioms. There are however a few axioms that cannot be written due to domain constraints, thus effectively reducing the number of expected axioms. We proceed by following the operation declaration order in Figure 4.1. First, observing with isEmpty. Easy: a empty tree is empty, a compound tree is not.

```
isEmpty(empty());
not isEmpty(make(X, T1, T2));
```

Next, observing with root. What is the root of an empty tree? This is left undefined since, as set in the **domains** section, operation root does not need to be

defined for empty trees. We are left with one axiom, the one that talks about the root of a compound tree.

```
root(make(X, T1, T2)) = X;
```

The cases for the left subtree and the right subtree are similar: such operations are not defined for empty trees. We then expect one axiom for each operation.

```
leftSubtree(make(X, T1, T2)) = T1;
rightSubtree(make(X, T1, T2)) = T2;
```

Now for the axioms related to the remaining, the **others**, axioms. We start with isLeaf. When is a tree a leaf? When it is not empty, and both the left and the right subtrees are empty. We write this all as follows.

```
isLeaf(T) iff not isEmpty(T) and
  isEmpty(leftSubtree(T)) and
  isEmpty(rightSubtree(T));
```

We can say the exact same thing in different words: an empty tree is not a leaf; a compound tree is a leaf is both subtrees are empty.

```
not isLeaf(empty());
isLeaf(make(X, T1, T2)) iff isEmpty(T1) and isEmpty(T2);
```

Operations classified as **others** allow extra flexibility in axiom writing. One can write on the left side of the equality symbol a term of the form isLeaf(T), or else an "observing a constructor" term.

The height of a tree is defined by induction: an empty tree has height 0; a compound tree has height 1 plus the maximum of the heights of the subtrees. Notice the primitive **max** function on integer values.

```
height(empty()) = 0;
height(make(X, T1, T2)) =
  1 + max(height(T1), height(T2));
```

Our last operation determines whether a tree is balanced: an empty tree is balanced; a compound tree is balanced when the height of its subtrees differ at most by one, and the subtrees are themselves balanced. We write all this as follows. Notice the primitive absolute value function **abs** on integer values.

```
isBalanced(empty());
isBalanced(make(X, T1, T2)) if
  isBalanced(T1) and isBalanced(T2) and
  abs(height(T1) - height(T2)) <= 1;
```

4.2 Refining Binary Tree into an Immutable Class

The refinement is in Figure 4.2. We have considered that the sort BinaryTree[Element] gets refined into the generic type LinkedBinaryTree<E>, and that sort Element is refined into type variable E.

We have decided to refine binary trees into an immutable class. This means that non-constructor operations whose return sort is a BinaryTree[Element] will become methods whose return type is LinkedBinaryTree<E>, rather than **void** as we have done in all the previous chapters. In this respect, the leftSubtree and the rightSubtree methods return new trees, rather than, for example, changing the target object state so as to keep the left(or the right) subtree only. When one looks at a Java signature of the form

```
LinkedBinaryTree<E> leftSubtree();
```

we immediately assume what we have just said: that the left tree is returned and that the target object remains unchanged. But that is not the only possible interpretation. In fact one can think of an implementation that records the left tree in the target object (thus, yielding a mutable class), and that returns something else (for example, the old tree). In order to distinguish the two interpretations, the keywords **return** and **this** can be used.

When the result of the specification operation is to be reflected on the state of the target object but the method happens to return an object of the same type, the keyword **this** needs to be used.

```
leftSubtree: BinaryTree[Element] -->? this:BinaryTree[Element]
      is  MyBinaryTree[Element] leftSubtree();
```

When, on the contrary, the result of the specification operation is to be returned by the method, no keyword is required. However, as in the case of Figure 4.2, we can make this explicit by using the keyword **return**.

```
leftSubtree: BinaryTree[Element] -->? return: BinaryTree[Element]
    is  LinkedBinaryTree[Element] leftSubtree();
```

Exercise 5 *Add a* size *operation to the* BinaryTree[Element] *specification in Figure 4.1. Can you simplify the axiom for* isLeaf*?*

Exercise 6 *Add a* occurences *operation that yields the number of times a given element occurs in a tree.*

Exercise 7 *Add one operation to determine whether a given tree is full and another to check if a tree is complete.*

```
specification BinaryTree[Element]
  sorts
    BinaryTree[Element]
  constructors
    empty: --> BinaryTree[Element][Element];
    make: Element BinaryTree[Element] BinaryTree[Element]
          --> BinaryTree[Element];
  observers
    isEmpty: BinaryTree[Element];
    root: BinaryTree[Element] -->? Element;
    leftSubtree: BinaryTree[Element] -->? BinaryTree[Element];
    rightSubtree: BinaryTree[Element] -->? BinaryTree[Element];
  others
    isLeaf: BinaryTree[Element];
    height: BinaryTree[Element] --> int;
    isBalanced: BinaryTree[Element];
  domains
    T: BinaryTree[Element];
    root(T) if not isEmpty(T);
    leftSubtree(T) if not isEmpty(T);
    rightSubtree(T) if not isEmpty(T);
  axioms
    T1, T2: BinaryTree[Element]; X: Element;
    isEmpty(empty());
    not isEmpty(make(X, T1, T2));
    root(make(X, T1, T2)) = X;
    leftSubtree(make(X, T1, T2)) = T1;
    rightSubtree(make(X, T1, T2)) = T2;
    isLeaf(T) iff not isEmpty(T) and
      isEmpty(leftSubtree(T)) and
      isEmpty(rightSubtree(T));
    height(empty()) = 0;
    height(make(X, T1, T2)) =
      1 + max(height(T1), height(T2));
    isBalanced(empty());
    isBalanced(make(X, T1, T2)) if
      isBalanced(T1) and isBalanced(T2) and
      abs(height(T1) - height(T2)) <= 1;
end specification
```

Figure 4.1: A specification for a binary tree of arbitrary elements.

```
refinement<E>
    E is  Element
    BinaryTree[Element] is LinkedBinaryTree<E> {
      empty: --> BinaryTree[Element] is LinkedBinaryTree();
      make: e:Element l:BinaryTree[Element]
        r:BinaryTree[Element] --> BinaryTree[Element] is
        LinkedBinaryTree(E e, LinkedBinaryTree<E> l,
                  LinkedBinaryTree<E> r);
      root: BinaryTree[Element] -->? Element is E data();
      leftSubtree: BinaryTree[Element] -->?
                  return:BinaryTree[Element] is
        LinkedBinaryTree<E> leftSubtree();
      rightSubtree: BinaryTree[Element] -->?
                  return:BinaryTree[Element] is
        LinkedBinaryTree<E> rightSubtree();
      isEmpty: BinaryTree[Element] is boolean isEmpty();
      isLeaf: BinaryTree[Element] is boolean isLeaf();
      height: BinaryTree[Element] --> int is int height();
      isBalanced: BinaryTree[Element] is boolean isBalanced();
    }
end refinement
```

Figure 4.2: A refinement mapping for the binary tree module.

Chapter 5

Sorted Sets

This chapter describes Sorted Sets of elements of a type that defines a total order. The Sorted Sets we are interested in provide the following operations.

empty to build an empty set;

insert to insert an element in a set;

remove to remove a given element from the set;

isIn to check whether the element is in the set;

isEmpty to check whether the set is empty;

largest to obtain the *largest* element in the set.

5.1 Building a Specification

Figure 5.2 presents a parameterised specification for a Sorted Set. This time the parameter is not the specification Element since we need to impose restrictions on the type of elements of Sorted Sets: we need to be able to compare two elements and we need that the underlying relation be a total order.

The parameter of our specification of Sorted Sets is the specification TotalOrder presented in Figure 5.1. This specification introduces the sort Orderable (which illustrates that the name of the sort does not need to be equal to that of the specification) and a single predicate qeq (greatest or equal) that represents a total order. The axioms of the specification specify the properties of a total order: transitivity, antisymmetry, and totality Additionally, axiom qeq(E, F) **if** E = F explicitly specifies that the relation is reflexive. Although this property is a consequence of

```
specification TotalOrder
  sorts
    Orderable
  others
    geq: Orderable Orderable;
  axioms
    E, F, G: Orderable;
    geq(E, F) if E = F;
    geq(E, F) if not geq(F, E);
    geq(E, G) if geq(E, F) and geq(F, G);
    E = F if geq(E, F) and geq(F, E);
end specification
```

Figure 5.1: A specification for a total order.

totality, since it is an important property relating equality and qeq, the decision was to make it explicit.

The classification of the five operations of Sorted Sets according to the three categories (**constructors**, **observers**, and **others**), and the **domains** sections are in the same line we have done before.

As usual, axioms are defined in terms of observations of the constructors. We have to describe the effect of a isEmpty, isIn, remove and largest operations, after a insert and the effect of a isEmpty, isIn operations, after a empty.

Let focus on the case of operation largest. We are looking for an axiom such as largest(insert(S, E)) = ... The easy case is when S is empty; the largest element of insert(S, E) is obviously E. When S is not empty, then we are looking for the largest element among E and those in S. But we know the largest element in S, it is obtained by largest(S). Then, the element we sought is the largest between E and largest(S). Taking advantage of the geq operation over elements of sort Orderable, we can write our axiom as follows.

```
largest(insert(S, E)) = E if isEmpty(S);
largest(insert(S, E)) = E
        if not isEmpty(S) and geq(E, largest(S));
largest(insert(S, E)) = largest(S)
        if not isEmpty(S) and not geq(E, largest(S));
```

The axioms for the other operations are similar to others we have seen before. Just notice that in the case of operation remove we have to take into account that the same element might have been inserted in the set more than once and hence it is necessary to propagate the insertion until reach the empty set.

More interesting and different are the last two axioms

```
insert(insert(S, E), F) = insert(S, E) if E = F;
insert(insert(S, E), F) = insert(insert(S, F), E);
```

They state two important properties of sets: order of insertions is irrelevant and insertion of an element already in the set does not change the set.

5.2 Sorted Set Module

As in previous cases, we must prepare a specification module. A directory containing the files named SortedSet.spc, TotalOrder.spc and a file containing the text in Figure 5.3 defines the module we need for the rest of this section.

5.3 Refining a Sorted Set

We consider a refinement of our module into class TreeSet, presented in Figure 5.4. In the refinement, presented in Figure 5.5, we have considered that the core specification SortedSet[TotalOrder] gets refined into the type SortedSet<E> and the parameter specification TotalOrder gets refined into the type variable E.

As shown in Figure 5.4, the class TreeSet defines a bound for its type parameter T with T **extends** IOrderable<T>. In this way, instantiations of TreeSet type are limited to classes C that implement IOrderable<C>. Such classes will necessarily have an implementation of the method **boolean** greaterEq(C e). The fact that the operation geq of TotalOrder corresponds to this method is specified as follows:

```
geq: Orderable e:Orderable is boolean greaterEq(E e );
```

In terms of refinement of operations into methods notice that we have

```
remove: SortedSet[Orderable] e : Orderable is
                                        boolean remove(E e);
```

It happens that method remove returns a boolean (possibly signaling whether the set has changed). No specification exists for this value (not covered by the specification) and, hence, for monitoring purpose is ignored.

```
specification SortedSet[TotalOrder]
  sorts
    SortedSet[Orderable]
  constructors
    empty: --> SortedSet[Orderable];
    insert: SortedSet[Orderable] Orderable
        --> SortedSet[Orderable];
  observers
    isEmpty: SortedSet[Orderable];
    isIn: SortedSet[Orderable] Orderable;
    remove: SortedSet[Orderable] Orderable
        --> SortedSet[Orderable];
    largest: SortedSet[Orderable] -->? Orderable;
  domains
    S: SortedSet[Orderable];
    largest(S) if not isEmpty(S);
  axioms
    E, F: Orderable;
    S: SortedSet[Orderable];

    isEmpty(empty());
    not isEmpty(insert(S, E));

    not isIn(empty(), E);
    isIn(insert(S,E), F) iff E = F or isIn(S, F);

    remove(empty(), E) = empty();
    remove(insert(S, E), F) = remove(S,E) when E = F
        else insert(remove(S,F), E);

    largest(insert(S, E)) = E if isEmpty(S);
    largest(insert(S, E)) = E
        if not isEmpty(S) and geq(E, largest(S));
    largest(insert(S, E)) = largest(S)
        if not isEmpty(S) and not geq(E, largest(S));

    insert(insert(S, E), F) = insert(S, E) if E = F;
    insert(insert(S, E), F) = insert(insert(S, F), E);

end specification
```

Figure 5.2: A specification for a Sorted Set.

```
module
   core
      SortedSet
   parameter
      TotalOrder
end module
```

Figure 5.3: A specification module for a Sorted Set.

```
public class TreeSet<T extends IOrderable<T>>
                                        implements Cloneable {
        public TreeSet() { ... }
        public boolean isEmpty() { ... }
        public boolean isIn(T item) { ... }
        public T largest() { ... }
        public void insert(T item) { ... }
        public boolean remove (T item) { ... }
        ...
}

//A type with a total order
public interface IOrderable<T> {
    boolean greaterEq(T e);
}
```

Figure 5.4: A Java generic class implementing a Sorted Set.

```
refinement <E>

  TotalOrder is E {
    geq: Orderable e:Orderable is boolean greaterEq(E e);
  }

  SortedSet[TotalOrder] is TreeSet<E> {
    empty: --> SortedSet[Orderable]
        is TreeSet();
    insert: SortedSet[Orderable] e:Orderable
            --> SortedSet[Orderable]
                is void insert(E e);
    isEmpty: SortedSet[Orderable]
        is boolean isEmpty();
    isIn: SortedSet[Orderable] e:Orderable
        is boolean isIn(E e);
    remove: SortedSet[Orderable] e:Orderable
        is boolean remove(E e);
    largest: SortedSet[Orderable] -->? Orderable
        is E largest();
  }

end refinement
```

Figure 5.5: A refinement mapping for the SortedSet module.

Chapter 6

Priority Queues

This chapter describes priority queues of elements of a type that can be compared with respect to their priority. The priority queues we are interested in provide the same sort of operations as the queues in Chapter 3, but with different meanings for operations dequeue and front. We are interested in the following operations.

make to build an empty queue;

offer to insert an element in a queue;

isEmpty to check whether the queue is empty;

minimum to obtain an element in the queue with the *lower* priority;

removeMinimum to remove the *minimum* element from the queue.

6.1 Building a Specification

Figure 6.2 presents a parameterised specification for a priority queue. This time the parameter is not the specification Element since we need to impose restrictions on the type of elements of priority queues: we need to be able to compare two elements in what concerns their priority. Contrarily to what happens in the case of sorted sets, we might have different elements with the same priority. Hence, what we need is that the type of elements of priority queues have a total pre-order to compare the priority of any two elements.

The parameter of our specification of priority queues is TotalPreOrder presented in Figure 6.1. This specification introduces the sort POrderable and a single

```
specification TotalPreOrder
  sorts
    POrderable
  others
    geq: POrderable POrderable;
  axioms
    E, F, G: POrderable;
    geq(E, F) if E = F;
    geq(E, F) if not geq(F, E);
    geq(E, G) if geq(E, F) and geq(F, G);
end specification
```

Figure 6.1: A specification for a total pre-order.

predicate qeq (greatest or equal) that represents a total pre order. The axioms of the specification specify the properties of a total pre order: transitivity and totality. As in the case of specification TotalOrder we explicitly also state reflexivity.

The classification of the five operations of priority queues according to the three categories (**constructors**, **observers**, and **others**), and the **domains** sections are as for simple queues (refer to Section 3.1 for details), except that dequeue is now called removeMinimum and front is called minimum.

We start, as usual, by observing the constructors. Following the discussion in Section 3.1, we are looking for four axioms. The two axioms for isEmpty are as for queues; all there remains are the axioms that describe the effect of a minimum and of a removeMinimum operation, after an offer.

For minimum we are looking for an axiom such as minimum(offer(Q, E)) = ... The easy case is when Q is empty; the smallest element of offer(Q, E) is obviously E. When Q is not empty, then we are looking for the smallest element among E and those in Q. But we know the smallest element in Q, it is obtained by minimum(Q). Then, the element we sought is the smallest between E and minimum(Q). Taking advantage of the geq operation in POrderable, we can write our axiom as follows.

```
minimum(offer(Q, E)) = E
        if not isEmpty (Q) and geq(minimum(Q), E) and
        not geq(E,minimum(Q));
minimum(offer(Q, E)) = minimum(Q)
        if not isEmpty(Q) and geq(E, minimum(Q)) and
        not geq(minimum(Q), E);
```

When minimum(Q) and E have the same priority the result of minimum(offer(Q, E))

must be either minimum(Q) or E. This means that we are in presence of an ADT that leaves some operations underspecified. We can express this as follows:

```
minimum(offer(Q, E)) = minimum(Q)
        if not isEmpty(Q) and geq(minimum(Q), E) and
        geq(E, minimum(Q)) and
        minimum(offer(Q, E)) != E;
minimum(offer(Q, E)) = E
        if not isEmpty(Q) and geq(minimum(Q), E) and
        geq(E, minimum(Q)) and
        minimum(offer(Q, E)) != minimum(Q);
```

Since the result of minimum(offer(Q, E)) is left underspecified we do not have means to specify the operation removeMinimum without introducing an auxiliary operation. In the specification presented in Figure 6.2 we opted for only including the axiom for a simple case and left the operation underspecified.

Another option would be to include in the specification an auxiliary operation that supports an additional way of observing the priority queue. For instance, we could include an auxiliar operation

```
//auxiliary
els:   PriorityQueue[POrderable] --> Bag[POrderable];
```

that observers the multiset of elements that are in a priority queue. For this, we would use the specification Bag that represents the multi set ADT. This specification is presented in Figure 6.3.

This operation needs to be marked as an auxiliary operation: it is for specification purpose only and, hence, not to be used by the clients of a priority queue. Since the specification language does not provide support for this, we just use a comment.

We could then use this observer to state that the multi set of elements we observe after applying removeMinimum is that obtained by removing that element chosen by the minimum operation, as follows:

```
els(removeMinimum(Q)) = remove(els(Q), minimum(Q))
            if not isEmpty(Q);
```

The operation itself would be specified as follows:

```
els(make()) = makeBag();
els(offer(Q, E)) = add(els(Q), E);
```

6.2 Priority Queue Module

As in previous cases, we must prepare a specification module. A directory containing the files named PriorityQueue.spc, TotalPreOrder.spc and a file containing the text in Figure 6.4 defines the module we need for the rest of this section. If we want to also have the auxiliary operation els in our specification, we would also need to include the specification Bag in our module.

6.3 Refining a Priority Queue into a Heap

We consider a refinement of our module into class HeapMinPriorityQueue, presented in Figure 6.5. In the refinement, presented in Figure 6.6, we have considered that the core specification PriorityQueue[TotalPreOrder] gets refined into the type HeapMinPriorityQueue<E> and the parameter specification TotalPreOrder gets refined into the type variable E.

As shown in Figure 6.5, the class HeapMinPriorityQueue defines a bound for its type parameter E with E **extends** IPOrderable<E>. In this way, instantiations of HeapMinPriorityQueue type are limited to classes C that implement IPOrderable<C>. Such classes will necessarily have an implementation of the method **boolean** greaterEq(C e). The fact that the operation geq of TotalPreOrder correspond to this method is specified as follows:

```
geq: POrderable e:POrderable is boolean greaterEq(E e );
```

Notice that although syntactically IPOrderable<E> is equal to the interface IOrderable<E> they have different semantics: implementations of method greaterEq<E> in classes that implement IOrderable<E> need to ensure anti-symetry, which is not required in classes that only implement IPOrderable<E>.

Exercise 8 *Write a specification of real priority queues, i.e., priority queues that work like the priority checkout lines in supermarkets, i.e., that preserve the "arrival" order of the elements with the same priority. When all elements are of the same priority, such priority queues must behave as queues in Figure 3.1.*

```
specification  PriorityQueue[TotalPreOrder]
  sorts
    PriorityQueue[POrderable]
  constructors
    make: --> PriorityQueue[POrderable];
    offer: PriorityQueue[POrderable] POrderable -->
                                        PriorityQueue[POrderable];
  observers
    isEmpty: PriorityQueue[POrderable];
    minimum: PriorityQueue[POrderable] -->? POrderable;
    removeMinimum: PriorityQueue[POrderable] -->?
                                        PriorityQueue[POrderable];
  domains
    Q: PriorityQueue[POrderable];
    minimum(Q) if not isEmpty(Q);
    removeminimum(Q)  if not isEmpty(Q);
  axioms
    Q: PriorityQueue[POrderable];  E: POrderable;
    isEmpty(make());
    not isEmpty(offer(Q, E));

    minimum(offer(Q, E)) = E if isEmpty(Q);
    minimum(offer(Q, E)) = E if not isEmpty(Q) and
            geq(minimum(Q), E) and
            not geq(E,minimum(Q));
    minimum(offer(Q, E)) = minimum(Q)
            if not isEmpty(Q) and geq(E, minimum(Q)) and
            not geq(minimum(Q), E);
    minimum(offer(Q, E)) = minimum(Q)
            if not isEmpty(Q) and geq(minimum(Q),E) and
            geq(E,minimum(Q)) and
            minimum(offer(Q, E)) != E;
    minimum(offer(Q, E)) = E
            if not isEmpty(Q) and geq(minimum(Q),E) and
            geq(E,minimum(Q)) and
            minimum(offer(Q, E))!= minimum(Q);

    removeMinimum(offer(Q, E)) = make() if isEmpty(Q);
end specification
```

Figure 6.2: A specification for a priority queue.

```
/**
 * The ADT specification for a Bag of elements
 * A bag of elements, providing for:
 *              adding and removing elements to/from a bag;
 *              checking whether the bag contains a given element.
 */
specification Bag[Element]
    sorts
        Bag[Element]
    constructors
        makeBag: --> Bag[Element];
        add: Bag[Element] Element --> Bag[Element];
    observers
        remove: Bag[Element] Element --> Bag[Element];
        howMany: Bag[Element] Element --> int;
        isEmptyBag: Bag[Element];
    others
        contains: Bag[Element] Element;
    axioms
        B: Bag[Element];
        E, F: Element;

        remove (makeBag(), E) = makeBag();
        remove (add (B, E), F) = B when E = F
            else add(remove(B, F), E);

        howMany(makeBag(), E) = 0;
        howMany(add(B, E), F) = 1 + howMany(B,F) when E = F
            else howMany(B, F);

        isEmptyBag(makeBag());
        not isEmptyBag(add(B, E));

        contains(B, E) iff howMany(B, E) > 0;

        add(add(B, E), F) = add(add(B, F), E);
end specification
```

Figure 6.3: A specification for a multiset.

```
module
  core
    PriorityQueue
  parameter
    TotalPreOrder
end module
```

Figure 6.4: A specification module for a priority queue.

```
public class HeapMinPriorityQueue<E extends IPOrderable<E>>
                                        implements Cloneable {
    public HeapMinPriorityQueue() { ...}
    public boolean isEmpty() { ...}
    public  E minimum(){ ...}
    public void add(E item){ ...}
    public void delMinimum(){ ...}

}
//A type with a total pre-order
public interface IPOrderable<E> {
    boolean greaterEq(E e);
}
```

Figure 6.5: A Java generic class implementing a priority queue with a min heap.

```
refinement<E>

  TotalPreOrder is E {
    geq: POrderable e:POrderable
         is boolean greaterEq(E e);
  }

  PriorityQueue[TotalPreOrder] is HeapMinPriorityQueue<E> {
    make: --> PriorityQueue[POrderable]
          is HeapMinPriorityQueue();
    offer: PriorityQueue[POrderable] e:POrderable -->
         PriorityQueue[POrderable]  is void add(E e);
    minimum: PriorityQueue[POrderable] -->? POrderable
         is E minimum();
    removeMinimum: PriorityQueue[POrderable] -->?
         PriorityQueue[POrderable] is void delMinimum();
    isEmpty: PriorityQueue[POrderable]
         is boolean isEmpty();
  }
end refinement
```

Figure 6.6: A refinement mapping for the PriorityQueue module.

Chapter 7

Maps

This chapter deals with maps from an arbitrary sort of keys into an arbitrary sort of values. We are interested in the following operations.

contains to check if there is a value associated with a key;

get to obtain the value associated with a key;

isEmpty to check whether the map is empty;

put to add an entry (a key-value pair) to the map;

remove to erase the entry for a given key, if present;

make to build an empty map.

7.1 Building a Specification

Figure 7.1 presents a parameterised specification for a map associating keys to values. It is parameterised by two specifications, Key and Value. Since there are no restrictions on the type of keys and values that can be used, these specifications only declare a sort.

We start by classifying the operations in the three available categories. Analysing the signatures derived from the above description, we obtain three candidates for constructors: make, put, and remove. Our very first question is, as usual, do we need them all, or is one (or more) operation redundant? In the case of the Map, does operation remove yield a map that would not be obtainable otherwise? If we want to remove an entry (that is, a key-value pair) from the map, it must have been placed there by a put operation, hence the map resulting from the remove operation can be obtained differently, by not performing the put in the first place.

The remaining four operators are all observers, for there seems to be no means of specifying, for example, what we mean by removing an element based on the remaining operations.

We are looking for 4×2 axioms, since we have three observers and two constructors. The axioms for isEmpty are simple, and very much like those we obtained for stacks and queues.

We turn our attention to the get operation. What is the result of asking for the value associated to a given key on an empty map? An error? Some agreed upon value? Do we have to be specific about it? We decide to declare get operation as partial. In the **domains** section, we specify that we can get the value associated to a key whenever there is a value associated:

```
get(M, k)  if contains(M, k)
```

In this way, the result of asking for the value associated to a given key on an empty map is *underspecified*, it might even not exist. We leave to the programmer that implements our specification to decide what to do in this case.

Now for the non-empty case. When looking for a key K1 on a map M where we have just added the entry (K, V), two cases arise: either K1 is K (in which case V is our value), or K1 is different from K (in which case we have to keep looking in M). We obtain the following equation.

```
get(put(M, K, V), K1) = V when K = K1 else get(M, K1);
```

As for removing the value associated with a key from a given map, if the map is empty, we simply ignore the request.

```
remove(make(), K) = make();
```

When removing K1 from a map M where we have just added the entry (K, V), we consider two cases: when K1 is and is not K. In either case we have to keep removing other K-entries in M, as accounted by the term remove (M, K1). When key K1 is not K, we keep looking for the key in M. And we must not forget to put back the entry (K, V) in the resulting map. The reason why we keep removing, even when K1 is K is because there may be other previously added K-entries in M. Notice that new K-entries replace old K-entries, only in the sense that operation get obtains the lastly added entry, ignoring others that may have been put in the map. Think, for example, of removing key K from the map M = put(put(make(),K,V1),K,V2). We want remove(M, K) to yield make() and not put(make(),K,V1). We obtain the following equation.

```
remove(put(M, K, V), K1) =
  remove(M, K1) when K = K1
  else put(remove(M, K1), K, V);
```

7.2 When are Two Maps Equal?

Are all maps freely generated from the constructors different? Does the insertion order matters? For example do the two terms, put(put(M,K,V),K1,V1) and put(put(M,K1,V1),K,V) denote two different maps when the keys K and K1 are different? and when they are equal?

In general we are not interested in the insertion order *for different keys*. We then add an axiom, relating constructors, to further constrain the maps generated from the constructors. (We have already met such an axiom in Section **??**, where we say that the order by which values are entered in a priority queue is not relevant.)

```
put(put(M, K, V), K1, V1) =
  put(put(M, K1, V1), K, V) if not K = K1;
```

When the keys are the same, the equation above does not apply. In fact we want the second insertion to overwrite the first. We use another axiom relating constructors to say exactly this:

```
put(put(M, K, V), K1, V1) = put(M, K, V1) if K = K1;
```

There is an alternative to further constrain the maps generated from the constructors: using the observers to specify when two sets are equal. When are two generic maps equal? When they have the same keys and, in this case, when the values associated with a same key are equal.

Suppose the have a predicate containsKey (cf. Exercise 9):

```
containsKey(M, K);
```

Then we can replace the two put/put axioms above by:

```
M = N iff (containsKey(M, K) iff containsKey(N, K) and
  get(M, K) = get(N, K));
```

7.3 A More Concrete Map

In the previous section we left underspecified the result of a get operation on an empty map. Suppose now that we want to be more specific in this case. We could take the view that one can only get (M, K), after making sure that the K is in M. We could setup a predicate containsKey, and always check containsKey before get. That would lead to rather inefficient implementations, if we think that, in general, it costs as much to check whether a key is in the map, as to obtain the value associated to the key.

What we would like is a means to signal that there is no value associated to a given key. In order to do so, we distinguish a special value in sort Value and call it noSuchKey. Values in maps are very much like elements in stacks (Figure **??**), only that they count with a special, noSuchKey, constructor, as follows.

```
noSuchKey: —> Value;
```

A specification for map values is in Figure 7.2. The specification for keys is that of the elements in stacks or queues (Figure **??**). In this case, looking for the value associated with a given key on an empty map can only result in the noSuchKey value.

```
get (make (), K) = noSuchKey ();
```

Finally, for the domains of our operations, we have to think whether we want noSuchKey values in our maps. In this particular case, allowing a noSuchKey value in the table would lead into difficulties when trying to interpret a noSuchKey result to get (M, K). Is it the case that M does not contain K? or it is the case that M contains an entry (K, noSuchKey)? We decided not to allow noSuchKey values in, hence the following entry in the **domains** section.

```
put (M, K, V) if V != noSuchKey ();
```

When are two maps as in Figure 7.3 equal? We now have an alternative to write to obtain the exact same effect: using the observer get to specify when two maps are equal. We write it as follows.

```
M = N iff get(M, K) = get(N, K);
```

7.4 Refining the Map Specification

As in the previous examples, we have to refine the whole module, composed in this case by the specifications for maps (Figure 7.3), keys (Figure **??**) and values (Figure 7.2). The result is in Figure 7.4.

Keys are very much like the elements in stacks (in fact we have reused their specification); they become arbitrary Objects. Values, however, now have a particular construct, noSuchKey. We take the advantage of the **null** reference in Java to encode such a constructor. So, rather than refining the noSuchKey operation into a particular method producing an object of class Object, we refine it directly into the primitive refinement expression **null**, as follows.

```
Value is class Object {
  noSuchKey: —> Value is null;
}
```

Exercise 9 *Add to specification* Map *an operation* containsKey *to check whether a given map contains an entry with a given key.*

Exercise 10 *Add to specification* Map *an operation* containsValue *to check whether a given map contains an entry with a given value.*

Exercise 11 *Consider allowing* noSuchKey *values in the map. In this case, a value* noSuchKey *returned by a* get *operation does not necessarily indicate that the map contains no entry for the key; it is also possible that the map explicitly maps the key to* noSuchKey. *Prepare a* containsKey *operation that may be used to distinguish these two cases (this is the approach of* java. util .HashMap.*)*

```
specification Map[Key, Value]

    sorts
        Map[Key, Value]

    constructors
        make : --> Map[Key, Value];
        put : Map[Key, Value] Key Value --> Map[Key, Value];

    observers
        get : Map[Key, Value] Key -->? Value;
        remove : Map[Key, Value] Key --> Map[Key, Value];
        isEmpty : Map[Key, Value];
        contains : Map[Key, Value] Key;

    domains
        M: Map[Key, Value];
        K: Key;
        get(M, K) if contains(M,K);

    axioms
      M: Map[Key, Value]; K, K1: Key; V, V1: Value;

          get(put(M, K, V), K1) = V when K = K1 else get(M, K1);

          remove(make(), K) = make();
          remove(put(M, K, V), K1) = remove(M, K1) when K = K1
          else put(remove(M, K1), K, V);

          isEmpty(make());
          not isEmpty(put(M, K, V));

          not contains(make(),K);
          contains(put(M, K, V),K1) iff K=K1 or contains(M,K1);

          put(put(M, K, V), K1, V1) = put(M, K, V1) if K = K1;
          put(put(M, K, V), K1, V1) = put(put(M, K1, V1), K, V) if not (K = K1);

end specification
```

Figure 7.1: A specification for a map from keys into values.

```
specification
  sorts
    Value
  constructors
    noSuchKey: --> Value;
end specification
```

Figure 7.2: A specification for values in maps.

```
specification
  sorts
    Map
  constructors
    make: --> Map;
    put: Map Key Value -->? Map;
  observers
    remove: Map Key --> Map;
    get: Map Key --> Value;
    isEmpty: Map;
  domains
    M: Map;  K: Key;  V: Value;
    put (M, K, V) if V != noSuchKey ();
  axioms
    M: Map;  K, K1: Key;  V, V1, V2: Value;
    get(make(), K) = noSuchKey();
    get(put(M, K, V), K1) = V when K = K1 else get(M, K1);
    remove(make(), K) = make();
    remove(put (M, K, V), K1) =
      remove(M, K1) when K = K1
      else put(remove(M, K1), K, V);
    isEmpty(make());
    not isEmpty(put(M, K, V));
    put(put(M, K, V), K1, V1) = put(M, K, V1) if K = K1;
    put(put(M, K, V), K1, V1) =
      put(put(M, K1, V1), K, V) if not (K = K1);
end specification
```

Figure 7.3: A specification for a map from keys into possibly noSuchKey values.

```
refinement
  Key is class Object
  Value is class Object {
    noSuchKey: --> Value is null;
  }
  Map is class ArrayMap {
    make --> Map is ArrayMap();
    put: Map k:Key v:Value -->? Map is
      void put(Object k, Object v);
    get: Map k:Key --> Value is Object get(Object k);
    remove: Map k:Key --> Map is void remove(Object k);
    isEmpty: Map is boolean isEmpty();
  }
end refinement
```

Figure 7.4: A refinement binding for the Map module.

Appendix A

clone() and equals()

Our method relies crucially on methods equals() and clone(), the latter for *mutable classes only*. This section discusses the writing of these methods.

A.1 The clone() Method

We start with Bloch's advice.

> To recap, all classes that implement Cloneable should override clone with a public method. This public method should first call super.clone and then fix any fields that need fixing. Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned and replacing the references to these objects with references to the copies [1].

The question that remains to answer is how deep do I have to copy?

> *Copy as deep as the operations of your class change its structure.*

The simplest case is that of *immutable* data structures, for example, class LinkedBinaryTree in Chapter 4. In this case the method in class Object need not be overridden. If methods change the state of an object, then state has to be duplicated. A stack (Chapter 2) implemented with an array, usually comprises (at least) two attributes: an integer describing the number of elements in the stack, and an array containing the elements in the stack.

```
private int size;
private Object[] elements;
```

Suppose that our implementation is such that no method changes the values of the stack—this is by far the most common case. Then we need to duplicate the

array, but not its elements. Since we do implement clone we catch the exception declared by the Object super class.

```
public ArrayStack<E> clone() {
  try {
    ArrayStack<E>result = (ArrayStack<E>) super.clone();
    result.elements =elements.clone();
    return result;
  } catch (CloneNotSupportedException e) {
    throw new InternalError(e.toString());
  }
}
```

An immutable class, such as LinkedBinaryTree mentioned before does not need to state **implements** Cloneable or to provide a clone method. In this case ConGu does not clone objects before applying the contracts, saving time and space. It will probably issue a warning to make sure that the class in question is indeed immutable, for it cannot check this property.

A.2 The equals() Method

When are two objects equal? What properties should hold for my equals() method? Again we start with Bloch's advice for a high-quality equals() method [1]:

1. Use the == operator to check if the argument is a reference to this object.
2. Use the getClass method to check if the argument is of the correct type.
3. Cast the argument to the correct type.

We exemplify the method with an array implementation of a stack of Chapter 2.

```
public boolean equals(Object other) {
  return this == other ||
    this.getClass() == other.getClass() &&
    equalArrays((ArrayStack)other);
}
```

The question remains, "when are two ArrayStack objects equal?" In simple cases we continue with Bloch's recipe:

4. For each "significant" attribute in the class, check to see if that field of the argument matches the corresponding field of this object.

We proceed as follows, by comparing the attributes size, and then elements.

```
private boolean equalArrays(ArrayStack other) {
  if (this.size != other.size)
    return false;
  for (int i = 0; i < size; i++)
    if (!ArrayStack.equalReferences(
        this.elements[i], other.elements[i]))
      return false;
  return true;
}
private static boolean equalReferences (Object one,
                                        Object other) {
  return one == null ? other == null : one.equals(other);
}
```

The above methodology is good for "linear" structures where order matters. These include, apart from stacks, queues, lists, strings, and vectors. What about a priority queue implemented with a heap (Chapter 6)? Is a priority queue with heap [3,6,9] any different from another with heap [3,9,6]? Certainly not, they represent the same priority queue. In such cases we abandon item 4. above, and turn to a more general approach:

Two objects are equal if they cannot be distinguished by any observer.

We use the observers (in the specification sense) to iterate on clones of the two queues. Method equalQueues can be written as follows.

```
private boolean equalQueues(HeapMinPriorityQueue<E> other) {
 HeapMinPriorityQueue<E> first = this.clone();
 HeapMinPriorityQueue<E> second = other.clone();
 while (!first.isEmpty() && !second.isEmpty()) {
     if (!first.min().equals(second.min()))
         return false;
     first.delMin();
     second.delMin();
 }
 return first.isEmpty() && second.isEmpty();
}
```

Notice that this equals crucially relies on the correct implementation of clone(); for such classes the **catch** clause above is never exercised. The above strategy can be easily adapted for most classes whose observers allow iteration. The interesting thing about this form of iteration is that it is independent of any particular implementation. This generality has a price: in some cases more efficient implementations can be found. Notice however that the proposed equals for heaps is $\mathcal{O}(n)$.

There are cases where the observers alone cannot provide for iterating over the structure. Nevertheless, the above approach is still valid if the implementation provides an iterator.

Finally, do not forget to override hashCode() when you override equals(), in order not to violate the general contract for Object.hashCode().

Bibliography

[1] Joshua Bloch. *Effective Java: Programming Language Guide*. Sun Microsystems, Inc., 2001.