Verifying Java-KE Programs

A Small Case Study

Arnd Poetzsch-Heffter

July 22, 2014

Abstract

This report investigates the specification and verification of a simple list class. The example was designed such that it illustrates reasoning techniques for heap-manipulating methods.

1 Program

We study the following classes:

```
class Node {
  int elem;
  Node next;
}

class List {
  Node elems;

  boolean isempty() {
    res = (elems == null);
  }

  void add( int par ) {
    Node newNd;
    Node oldNd;
    newNd = new Node();
    newNd.elem = par;
    oldNd = this.elems;
    newNd.next = oldNd;
    this.elems = newNd;
  }

  void append( List par ) {
    Node appl;
    appl = par.elems;
    while( appl != null ) {
      int el;
      el = appl.elem;
      this.add( el );
      appl = appl.next;
    }
  }
}
```

2 Specification

The following method specifications need the store invariant invList and the abstraction function aList for lists. The invariant is defined such that it is

- maintained by all methods
- sufficiently strong to guarantee well-definedness of aList
- sufficiently strong to verify the method specifications

The helper predicate nListP checks whether a value is a Node-object that, in a given store, represents a given list:

```
nListP :: Value → Store → int list → bool
  nListP null OS []                 =  true
  nListP (ref Node nid) OS (x#xl) =  (OS((ref Node nid).elem) = x)
                                  ∧ nListP (OS((ref Node nid).next)) OS xl
  nListP _ _ _                      = false

lemma
nListP nr OS nl1 ∧ nListP nr OS nl2 ⟶ nl1 = nl2

invList :: Store → bool
  invList OS =  wts OS
   ∧ (∀ lr. lr = (ref List lid) ⟶ ∃al. nListP (OS(lr.elems)) OS al )

aList :: Value → Store → int list
 aList lr OS  = if ∃lid. lr = (ref List lid)
                    then  THE al. nListP (OS(lr.elems)) OS al
                    else  arbitrary
```

For method isempty, we specify three properties:

- It does not modify the heap:

  ```
  { OS = $ } List@isempty { OS = $ }
  ```

- It does not throw exceptions:

  ```
  { true }  List@isempty { exc = null }
  ```

- It checks that the this-object represents an empty list; more precisely, if the object store satisfies the store invariant invList for lists and the abstraction of *this* list is AL, then the result is true if AL is empty, and false otherwise.

  ```
  { (invList $) ∧ (aList this $)=AL } List@isempty { res = (AL=[]) }
  ```

For method add, we specify three properties:

- It maintains the store invariant:

  ```
  { invList $ } List@add { invList $ }
  ```

- It does not throw exceptions:

  ```
  { invList $ } List@isempty { exc = null }
  ```

- It adds the parameter as new first element to the list:

```
{ THIS=this ∧ PAR=par ∧ (aList this $)=AL ∧ invList $ }
List@add { (aList THIS $) = PAR#AL) }
```

- It maintains all functional properties of the store that are independent of the instance variable this.elems. This is the so-called *frame specification*:

```
{ (independ F {this.elems} $) ∧ (F $)=X ∧ invList $ }
List@add { (F $)=X }
```

The independence predicate can be formalized as follows:

```
independ :: (Store → a) → (InstVar set) → Store → bool
  independ f ivs OS = ∀ OS1.
     (∀ iv. iv ∉ ivs ∧ (alive iv OS) ⟶ OS(iv)=OS1(iv) ∧ (alive iv OS1) )
     ⟶ f OS = f OS1
```

For method append, we consider here only the specification of its functional behavior:

```
{ THIS = this ∧ par != null ∧ (aList this $) = AL ∧ (aList par $) = BL
∧ invList $ }
   List@append
{ (aList THIS $) = (rev BL)@AL ∧ exc = null ∧ invList $ }
```

3 Verification

In the following, we concentrate on the verification of the functional properties of method `append`. The proof illustrates the use of adaptation rules, abstraction functions, reasoning with frame properties, and an abstract loop invariant. The proof uses three helper functions:

- In stores satisfying the invariant, nList abstracts the list reachable from a Node-object:

```
nList :: Value → Store → int list
 nList nr OS = if ∃al.nListP nr OS al then THE al.nListP nr OS al
                                      else arbitrary
```

- A list yl1 is a *suffix* of a list yl2 if it can be extended at the front to yl2

```
isSuffix :: int list → int list → bool
  isSuffix yl1 yl2 = ∃ xl. xl@yl1 = yl2
```

- The list that does this described extension is the corresponding prefix:

```
cPrefix :: int list → int list → int list
  cPrefix yl1 yl2 = if isSuffix yl1 yl2 then THE xl. xl@yl1 = yl2
                                        else arbitrary
```

The proof of method `append` is outlined in Fig. 1. We first explain the strengthening steps (s1)-(s3) and the weaking step (w1). The proof of the invocation statement this.add(el) is given in Fig. 2 and will be explained afterwards. As we show that each statement terminates normally (exc=null), we can always use the second disjunct of the seq-rule. Basic to the proof is the loop invariant (the assertion (INV) before the while loop). It essentially states that

- the abstracted list reachable from variable appl is a suffix *CL* of the list *BL* reachable from par in the prestate of method append

```
  { THIS = this ∧ par != null ∧ (aList this $) = AL ∧ (aList par $) = BL
    ∧ invList $ }
  ⟶ (s1)
  { par != null ∧ exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList ($(par.elems)) $) BL))@AL
    ∧ isSuffix (nList ($(par.elems)) $) BL ∧ invList $}

appl = par.elems;

  { exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ invList $ } (INV)

while( appl != null ) {

  { appl != null ∧ exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ invList $ }
  ⟶ (s2)
  { appl != null ∧ exc = null ∧ THIS = this
    ∧ (aList this $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ $(appl.elem) = $(appl.elem) ∧ invList $ }

el = appl.elem;

  { appl != null ∧ exc = null ∧ THIS = this
    ∧ (aList this $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ el = $(appl.elem) ∧ invList $ }

this.add(el);

  { appl != null ∧ exc = null ∧ THIS = this
    ∧ (aList THIS $) = $(appl.elem)#(rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ invList $ }
  ⟶ (s3)
  { appl != null ∧ exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList ($(appl.next)) $) BL))@AL
    ∧ isSuffix (nList ($(appl.next)) $) BL ∧ invList $ }

appl = appl.next;

  { exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ invList $ }

}

  { (exc!= null ∨ appl = null) ∧ exc = null ∧ THIS = this
    ∧ (aList THIS $) = (rev (cPrefix (nList appl $) BL))@AL
    ∧ isSuffix (nList appl $) BL ∧ invList $ }
⟶ (w1)
  { (aList THIS $) = (rev BL)@AL ∧ exc = null ∧ invList $ }
```

Figure 1: Proof outline for the functional properties of `append`

- the abstracted list reachable from this is the reverse of the corresponding prefix of CL appended to the abstracted list AL reachable from this in the prestate.

To exploit properties of the list abstractions, we make sure that we only use them in contexts where the store invariant holds. Now, we consider the strengthening and weakening steps:

(s1) From (aList par $) = BL, we get (nList ($(par.elems)) $) = BL; thus, the corresponding prefix is empty, yielding the fourth conjunct.

(s2) Trivial

(s3) Here, we essentially use the property x#(rev yl) = rev (yl@[x]) and the fact that by taking the tail of the list reachable from appl, the corresponding prefix gets longer by the head.

(w1) Because of appl = null, the corresponding prefix is BL.

It remains to show that the annotation of the invocation statement this.add(el) is correct (see Fig. 2). We explain the proof from inside out. To obtain the innermost triple,

- apply the invocation rule to all specification triples of add, twice to the frame specification
- in one of the frame triples resulting from the frame specification, rename the logical variables (by the subst-rule)
- take the conjuction of the resulting triples

Step (a1) consists of

- substituting (λ OS. nList APPL OS) for F and (λ OS. OS(APPL.elem)) for G
- substituting $CL@AL$ for AL
- applying the invoc-var-rule to T and Z

Strengthening step (s4) makes use of β-reduction and shows the independence[1]. Weakening step (w2) makes use of β-reduction. Adaptation step (a2) uses the invar-rule to add state-independent properties to pre- and postcondition (the conjuncts only contain logical variables). Strengthening step (s5) uses the equations of the logical variables to refine some of the conjuncts. Weakening step (w3) is similar. In addition, it "forgets" the equations. Consequently, the logical variables CL, X, and PAR no longer appear in the postcondition. Thus, we can use the ex-rule to existentially quantify over these variables in the precondition (step (a3)). Now we can eliminate the equations for CL, X, and PAR also in the precondition using strengthening, obtaining the triple that we used in Fig. 1.

Q.E.D.

[1]The proof of independence needs typing information about this that can be obtained by strengthening (not shown here).

```
{ appl != null ∧ exc = null ∧ THIS = this
  ∧ (aList this $) = (rev (cPrefix (nList appl $) BL))@AL
  ∧ isSuffix (nList appl $) BL ∧ el = $(appl.elem) ∧ invList $ }
⟶ (s6)
{ ∃X,CL,PAR. appl != null ∧ exc = null ∧ THIS = this
  ∧ (aList this $) = (rev (cPrefix (nList appl $) BL)@AL
  ∧ isSuffix (nList appl $) BL ∧ el = ($(appl.elem)) ∧ invList $
  ∧ (nList appl $) = X ∧ appl=APPL ∧ CL = (rev (cPrefix X BL)
  ∧ PAR = el ∧ ($(appl.elem)) = PAR }
──────────────────────────────────────────────────────────────────────────────────────↓ (a3)
{ appl != null ∧ exc = null ∧ THIS = this
  ∧ (aList this $) = (rev (cPrefix (nList appl $) BL)@AL
  ∧ isSuffix (nList appl $) BL ∧ el = ($(appl.elem)) ∧ invList $
  ∧ (nList appl $) = X ∧ appl=APPL ∧ CL = (rev (cPrefix X BL)
  ∧ PAR = el ∧ ($(appl.elem)) = PAR }
⟶ (s5)
{ this != null ∧ THIS = this ∧ PAR = el ∧ (aList this $) = CL@AL
  ∧ (nList appl $) = X ∧ ($(appl.elem)) = PAR
  ∧ invList $ ∧ appl=APPL
  ∧ APPL != null ∧ isSuffix X BL ∧ CL = (rev (cPrefix X BL) }
──────────────────────────────────────────────────────────────────────────────────────↓ (a2)
{ this != null ∧ THIS = this ∧ PAR = el ∧ (aList this $) = CL@AL
  ∧ (nList appl $) = X ∧ ($(appl.elem)) = PAR
  ∧ invList $ ∧ appl=APPL }
⟶ (s4)
{ this != null ∧ THIS = this ∧ PAR = el ∧ (aList this $) = CL@AL
  ∧ (independ (λOS. nList APPL OS) {this.elem} $ )
  ∧ (λOS. nList APPL OS) $ = X
  ∧ (independ (λOS. OS(APPL.elem)) {this.elem} $)
  ∧ (λOS. OS(APPL.elem)) $ = PAR ∧ invList $ ∧ appl=APPL }
──────────────────────────────────────────────────────────────────────────────────────↓ (a1)
{ this != null ∧ THIS = T ∧ PAR = el ∧ (aList this $) = AL
  ∧ (independ F {this.elems} $) ∧ F $ = X
  ∧ (independ G {this.elems} $) ∧ G $ = PAR ∧ invList $ ∧ Z=APPL }

this.add(el);

{ THIS = T ∧ (aList THIS $) = PAR#AL ∧ exc = null
  ∧ F $ = X ∧ G $ = PAR ∧ invList $ ∧ Z=APPL }
──────────────────────────────────────────────────────────────────────────────────────↑ (a1)
{ THIS = this ∧ (aList THIS $) = PAR#(CL@AL) ∧ exc = null
  ∧ (λOS. nList APPL OS) $ = X ∧ (λOS. OS(APPL.elem)) $ = PAR
  ∧ invList $ ∧ appl=APPL }
⟶ (w2)
{ THIS = this ∧ (aList THIS $) = PAR#(CL@AL) ∧ exc = null
  ∧ (nList appl $) = X ∧ ($(appl.elem)) = PAR ∧ invList $ ∧ appl=APPL }
──────────────────────────────────────────────────────────────────────────────────────↑ (a2)
{ THIS = this ∧ (aList THIS $) = PAR#(CL@AL) ∧ exc = null
  ∧ (nList appl $) = X ∧ ($(appl.elem)) = PAR ∧ invList $ ∧ appl=APPL
  ∧ APPL != null ∧ isSuffix X BL ∧ CL = (rev (cPrefix X BL) }
⟶ (w3)
{ appl != null ∧ exc = null ∧ THIS = this
  ∧ (aList THIS $) = $(appl.elem)#(rev (cPrefix (nList appl $) BL))@AL
  ∧ isSuffix (nList appl $) BL ∧ invList $
──────────────────────────────────────────────────────────────────────────────────────↑ (a3)
{ appl != null ∧ exc = null ∧ THIS = this
  ∧ (aList THIS $) = $(appl.elem)#(rev (cPrefix (nList appl $) BL))@AL
  ∧ isSuffix (nList appl $) BL ∧ invList $ }
```

Figure 2: Proof outline for the triple of `add` used in proof of `append`