POPL2021 Endsem   Nov 30, 2021

NOTE:

1. Presenting your answers properly is your responsibility. You lose credit if you cannot present your ideas clearly, and in proper form. Please DO NOT come back for re-evaluation saying, "What I actually meant was . . . ".
2. Be precise and write clearly. Remember that somebody has to read it to evaluate!

---

1. Consider the following extension for the data constructor `Expr`:

```
data Expr   =  I Var | Add Expr Expr | Sub Expr Expr
               | IF BExpr Expr Expr | PP Var
data BExpr  =  GT Expr Expr | NOT BExpr
data Var    =  X | Y | Z
```

   `BExpr` denotes Boolean expression using Greater-than (GT, >) and Not (NOT) operations. `IF` represents if-then-else expression. `PP` stands for increment (`++`, as in `++x`). Other terms have meaning as used in the class.

   Extend the StateMonad discussed in the class to also count the number of additions and subtractions performed during evaluation (`eval`). Note that this can be thought of as a profiling interpreter for a program. Do not count the number of increments.

   Now state is a 3-tuple as follows:

```
type State = (Var → Int, Int, Int)
```

   The second and the third elements of the tuple denote the number of additions and subtractions respectively.

   Note that merely using the `State` type does not solve the purpose. You need to define the Monad operations (`return` and >>=) over extended StateMonad, and use these operations to implement `eval` function. (20)

2. Consider the Haskell expression below (The . is the composition function):

```
map (.) map
```

   i. Show all the pair of terms that need to be unified to infer the type for the expression. Show the MGU for each pair, wherever it exists. In case the MGU does not exist, explain the reason briefly. (15)

ii. As you may have noticed, some of the pairs cannot be unified. Modify the last term in the expression (the second occurrence of `map`) to fix the issue (so that type inference process goes through). Make minimal possible changes such that the `map` function remains present. (5)

iii. Show the modified pairs that need to be unified and the corresponding MGUs. (10)

3. Consider the following simple language for file I/O, described in an informal BNF form:

```
Program  :  Decls Stmts          // declarations followed by statements

  Decls  :  D                    // One or
         |  Decls D              // more declarations

  Stmts  :  S                    // One or
         |  Stmts S              // more statements

      D  :  INT v;               // Integer variable or
         |  FILE v;              // File variable declaration

      S  :  v.open();
         |  v.close();
         |  v1.write(v2);        // Writing INT v2 to a file v1
         |  v1 = v2.read();      // Reading INT value from a file v2 into INT v1
         |  v1 = v2;             // Only INT-s can be assigned
```

For simplicity, we assume that declaring a FILE variable associates it with with a file name on the disk, so we ignore parameters (such as the filename to open). Also, the grammar for variables (v) is not shown — you can assume C like variables.

Following semantic properties hold for the execution:

1. A variable should be declared only once.
2. ALL the files are CLOSED at the start of execution. They are also CLOSED automatically at the end of the execution, if not explicitly closed by the program.
3. Opening a file is COMPULSORY for performing any reads or writes; i.e. any reading and writing on a file that is not open will generate error.
4. Opening an already opened file is allowed.
5. Closing an open file will close it for read and write operations; i.e. any further reading and writing on the file will generate error. It does not matter how many times the file was opened.

6. Closing a closed file is NOT allowed.
7. Reading an open file returns an INT (integer). You can only write an INT to an open file.
8. An INT variable can be assigned to other INT variable, but FILE variable can not be assigned to other FILE variable.

The types for this language can be from T, where:

T	:	INT	// Integer variable
	|	FILE	// File variable
	|	Valid	// A semantically valid statement
	|	...	// Use any other type you may need

A program is Valid if all the constituent statements are Valid.

Design a type system to capture the constraints described above. In particular, you need to describe axioms and inference rules for various constructs above. Note that you will also need to manipulate the type environment (Γ) as part of the rules. (30)

4. Consider the *Optimistic Synchronization* discussed in the class to implement concurrent operations on the List-based Sets. The Sets support *add*, *remove*, and *search* operations only. The code for *add* and *remove* is reproduced from the book (See the next page).

   For each of the statement below, state whether it is True or False. Give brief justification for your answer (proof sketch or counter-example):

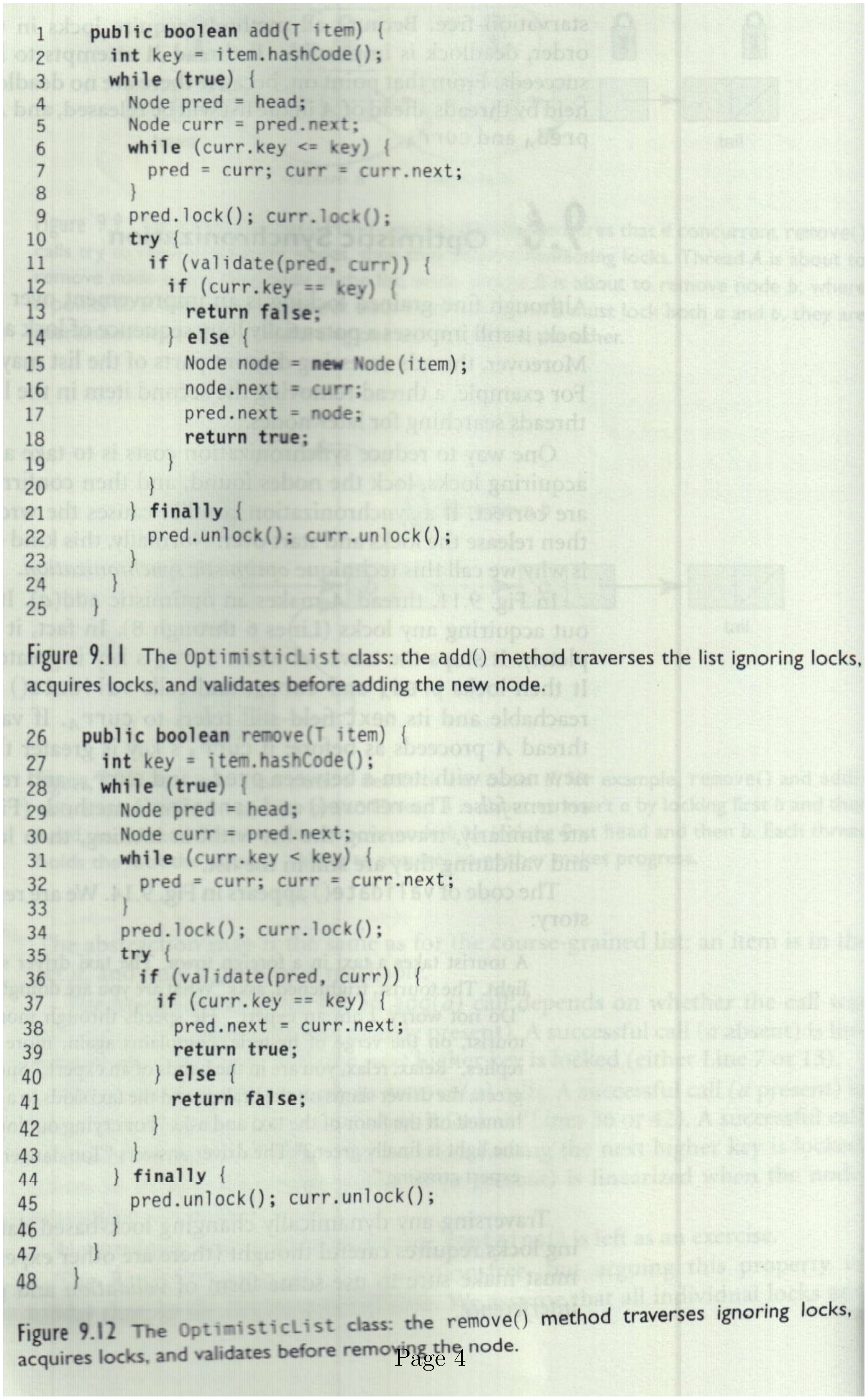
   i. Lock on *pred* is not required for the correctness of *add* (Fig 9.11; Line 9). (5)

   ii. Lock on *curr* is not required for the correctness of *add* (Fig 9.11; Line 9). (5)

   iii. Lock on *pred* is not required for the correctness of *remove* (Fig 9.12; Line 34). (5)

   iv. Lock on *curr* is not required for the correctness of *remove* (Fig 9.12; Line 34). (5)

(Image from the book reproduced on the next page.)



```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key <= key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Node(item);
          node.next = curr;
          pred.next = node;
          return true;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

Figure 9.11 The OptimisticList class: the add() method traverses the list ignoring locks, acquires locks, and validates before adding the new node.

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          pred.next = curr.next;
          return true;
        } else {
          return false;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

Figure 9.12 The OptimisticList class: the remove() method traverses ignoring locks, acquires locks, and validates before removing the node.