Programming with Undo

by

Ersoy Bayramoğlu

Submitted to the School of Computer and Communication Sciences

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

August 2009

Author . . . .

School of Computer and Communication Sciences

August 14, 2009

In the memory of my mother, Horise (1959-2008)

Programming with Undo

by

Ersoy Bayramoğlu

Submitted to the School of Computer and Communication Sciences
on August 14, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis is about objects that can undo their state changes. Based on an earlier work on data structure persistence [15], we propose generating undo methods for classes from annotated classes automatically. As opposed to ephemeral data structures, persistent data structures carry their older versions, and undo for a persistent structure is just returning to a previous version. Undoable objects simplify programming in a number of areas such as backtracking in constraint programming, and undo for interactive applications. Using the undo methods of individual objects, larger application level undo functionality can be built in an easier way.

Based on this thesis we have implemented a tool, which can produce Java classes that have undo methods from annotated Java source code files. Compared to the effort of implementing undo manually, the overhead that the number of annotations needed brings is not significant.

Thesis Supervisor: Viktor Kuncak

Thesis Supervisor: Adam Granicz

Acknowledgements

I am grateful to my advisor, Viktor Kuncak, for all his help and patience during the last two years. I learned immensely from him especially on verification and doing research. If I can ever become a researcher, he will have a big part in it. I thank my advisor at IntelliFactory, Adam Granicz, for giving me the opportunity to work in Budapest, and for his hospitality. I thank my office mates Joel Björnson, Diego Echeverri, and Anton Tayanovskyy for the discussions we had on life, functional programming, our home countries, and Estonia. (Yes, Estonia.) I will definitely miss our beer nights at Berliner. I thank my friend Serdar Kadıoğlu for his help on constraint programming, and always finding time to answer my questions despite the big time difference we had. The project originated from our initial discussions and his frustration of debugging backtracking code. Now that the time difference has vanished, I hope to work with him more. Finally, I thank Jóci, Ruben, Péter, Baráth, and Balázs for making my stay in Budapest much more enjoyable than it would have been without them. They will be missed.

Table of Contents

Chapter 1

Introduction

The undo operation, mostly popularized by text editors, goes back to the ENIAC [24] times. The undo facility was provided to return to previous points along the solution of a nonlinear differential equation on the ENIAC computer. Today we see undo in all kinds of editors such as text editors like emacs [30], graphics editors like CAD systems, and spreadsheets. Undo has many other important uses in areas like debugging, databases, programming language environments – such as Interlisp [32], COPE [7], and Cornell Program Synthesizer [31]-, fault tolerant systems, and error recovery. For instance, consider debugging a faulty program. Usually the errors cannot be pinpointed easily by just setting a break point, and in that case it may be desirable to go a few steps back without running the whole program all over again.

1.1 Undo Models

An undo model is generally defined by the combination of commands, history of these commands, and the undo operator that works to manipulate the history. (The history does not have to be there physically, but even in that case it is good to think in terms of a history.) Different undo models have been proposed over the years. One property that distinguishes different undo models is the representation of the undo operation itself. Some systems like the emacs editor [30], provide undo as a primitive operation, that is the undo operation itself is added to the history and can be undone. In this setting, there is no need for an explicit redo operation. Other systems like the COPE interactive program editor [7], provide undo as a meta-command. In this model, undo manipulates the history, but never appears in it. These systems typically have explicit redo meta-commands.

The second distinguishing feature is whether the undo model provides a linear undo operation or a selective [8] one. Linear undo can only undo the effects of the most recently performed operation in the history file. In order to undo the effects of another operation, the user has to undo all the operations that were performed after that. In a selective undo model, a user can undo the effects of an arbitrary operation in the history without dealing with the following operations. One important difference between different types of selective undo models is how they deal with dependencies between operations. Some models ignore the dependencies. COPE's script model [7] is one of them. Programs in the script model are a sequence of scripts. The user can selectively undo the effects of any script executed before. COPE does not undo the effects of the scripts that depend on the undone script. Other selective undo models advocate undoing the effects all of the subsequent operations that depend on the undone operation.

Undo can help users recover from their errors in interactive systems. Brown [10] argues that application level undo offers a very limited form of error recovery. It cannot account for problems like operator errors made during upgrades and reconfiguration, external viruses and hacker attacks. To address these issues he offers system wide undo, which can restore system-level state including operating system and application configuration, the executable binaries for the OS and application software, and software patches, libraries and modules.

1.2 Goals

Brown [10] defines an undoable system as any system that is fundamentally concerned with *state* and *time*, and that allows time to flow logically in reverse, such that previous versions of the state can be recreated or restored. Following this definition, we propose ***undoable objects*** that can restore previous versions of their states through automatically generated undo methods. The undo method is a linear one. Even though undo will be just an ordinary method, it will act as a meta-command, i.e. it will only undo the effects of other methods called on the object, not itself. As individual objects will carry undo methods, we call this undo model, ***object level undo***, and argue that it makes building application level undo easier.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 gives a quick overview on past research on persistence, which is the basis for our undoable objects. Chapter 3 introduces the programming model, and explains how to make a class persistent. Chapter 4 contains specific applications that benefit from undoable objects. Chapter 5 presents the related work in the area. Chapter 6 is concludes and also includes some ideas for future work.

Chapter 2

Background

One way of supporting undo for individual objects is by keeping the old values of the objects' fields, and restoring the old values whenever undo is performed. We will base our undoable objects on the previous research done on *persistent* data structures [21]. The initial configuration of a data structure is counted as version zero, and each update operation on the structure yields a new version. Persistent data structures allow access to all versions. Ordinary objects are *ephemeral*, i.e. an update on an object destroys the previous version, and only the newest version of the object remains. A straightforward, but an inefficient, way of accomplishing persistence would be taking a copy of the data structure each time one of its fields is modified. Driscoll, Sarnak, Sleator, and Tarjan [15] developed more efficient and systematic ways of making linked data structures persistent. A *linked data structure* is a finite collection of nodes, each containing a fixed number of named fields. Each field is either an information field, able to hold a single piece of information of a specified type, or a pointer field, able to hold a pointer to a node or the special value null indicating no node. General techniques for making arrays persistent were also developed [14].

2.1 Making Data Structures Persistent

A data structure is said to be *partially persistent* if all versions can be accessed but only the newest version can be modified. The easiest method to transform a linked data structure into its persistent form is the *fat node* method. The main idea is to allow nodes to become arbitrarily "fat," i.e., to hold an arbitrary number of values of each field. Each fat node will contain the same information and pointer fields as an ephemeral node (holding original field values), along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp. The version stamp indicates the version in which the named field was changed to have the specified value. In addition, each fat node has its own version stamp, indicating the version in which the node was created.

Ephemeral update steps on the fat node structure are simulated as follows. Consider update operation i. When an ephemeral update step creates a new node, a corresponding new fat node, with version stamp i, containing the appropriate original values of the information and pointer fields is created. When an ephemeral update step changes a field value in a node, the corresponding new value to the corresponding fat node are added, along with the name of the field being changed and a version stamp of i. For each field in a node, only one value per version is stored; when storing a field value, if there is already a value of the same field with the same version stamp the old value is overwritten. Original field values are regarded as having the version stamp of the node containing them.

The fat node brings an O(log n) overhead, where n is the number of versions, to access or modify a field of a particular node in a particular version. The authors of [15] proposed more efficient methods for data structures with nodes of bounded indegree. The *node copying* method produces persistent versions of such structures with O(1) amortized time overhead for access and update operations.

A data structure is said to be *fully persistent* if every version can be both modified and accessed. The lack of linear order makes the navigation in a fully persistent fat node data structure inefficient. Similar to the partial persistence case, for linked structures with nodes of bounded indegree, the *node splitting* method [15] produces fully persistent versions with O(1) amortized time overhead for access and update operations.

2.2 Selective Persistence

We want to be able to undo the state changes of objects. Fat nodes that store the older values of fields are ideal for this purpose. However, we do not need to access the older version of an object before performing undo. Therefore, both partial persistence, and full persistence is too strong for undoable objects. We use a more restricted persistence scheme that we call *selective persistence*. A data structure is said to be selectively persistent if only its newest version can be modified, and it only allows accessing older versions after performing a specified operation. In our case, the operation is undoing.

Fat nodes in our case are stacks, so that only the newest version of each field (top of the corresponding stack) can be accessed. The older versions can only be accessed when undo is performed and the top of the stacks are popped.

Another change is that we dropped the version numbers. They are problematic in a number of ways. First of all, if local, per object version numbers are used, then synchronization is needed whenever two persistent data structure are merged. If we use a global notion of time, then it is too restrictive, because integers are bounded. Also, it requires all persistent objects in the program to share a global field. Another problem is related to memory consumption, because each field will have an additional integer version number. Instead of using version numbers, we record the modified references for each individual step that will be undone later. (The notion of a step will be made precise in later chapters.) There is a trade-off here, because when a field is modified within a step, a look up on the history will be performed to see if it was modified within the same step. This look up operation is expensive.

Persistent arrays are problematic, because accessing the elements of an array takes O(log log m) time [21], where m is the number of updates on an array. We do not want an overhead greater then O(1) time for access. Selectively persistent arrays give us precisely that. We replace the array elements of type `T` with type `Stack<T>`. Therefore `array[T]` is implemented as `Stack<array[stack<T>]>`. The outer stack is to keep the modifications on the array reference itself, and the inner one is for the elements of the array.

Chapter 3

Programming Model & Implementation

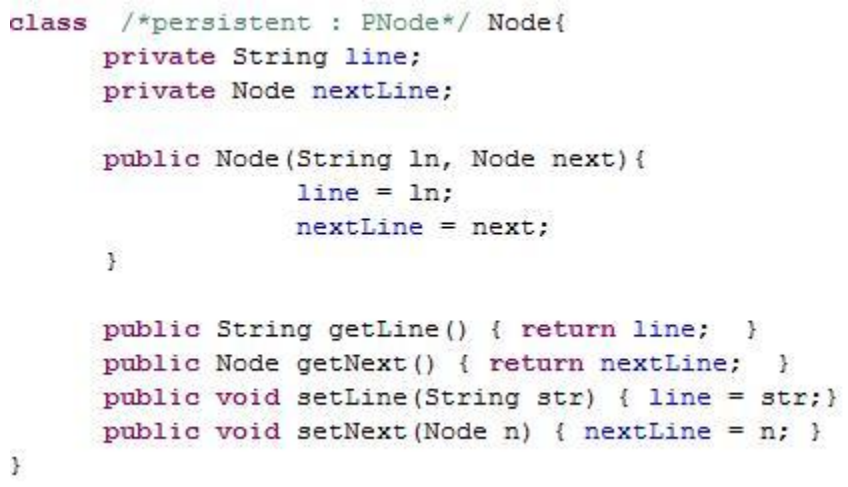
The programming model is very simple and intuitive. It requires a few annotations to generate the persistent version of the ephemeral classes. After the generation step, the new classes can be used like the old ones. Additionally, they provide the undo method to go back to the previous versions.

We use fat nodes to store old values of fields. The fat nodes in this case will be stacks. Also, each undoable data structure will contain a history stack. Each element of the history stack will be a hash set that contains references to the fields modified in an individual step. A step is the unit of undo, and for now it can be thought as a method call, i.e. a method call on an object pushes a new entry in its history stack and all the changes in the duration of that method call will be recorded in that entry. This definition of a step is too restrictive, and later I will talk about how programmers can adjust it according to their needs. Since all (undoable) fields of an undoable structure are stacks, the type of the history field will be `Stack<HashSet<Stack>>`. Undoable structures may contain ephemeral fields that are not stacks. In that case, the changes on them will not be stored, and they will not participate in the undo operation. The undo method will simply traverse the top of the history stack, and pop the top of the hash set's elements.

3.1 A Persistent Linked List Class

I will use a Linked List class to demonstrate how ephemeral data structures will be transformed into persistent ones containing an undo method. In my first attempt I considered making only the list node class persistent. Then individual nodes would have histories, and the changes on the nodes would be recorded in themselves. However, this is not very useful, because to undo changes, the undo method would have to be called on individual nodes. In order to undo methods called on a linked list object, we have to make the linked list class persistent too. Therefore the only history we need in this case is for the whole list, not for individual nodes.

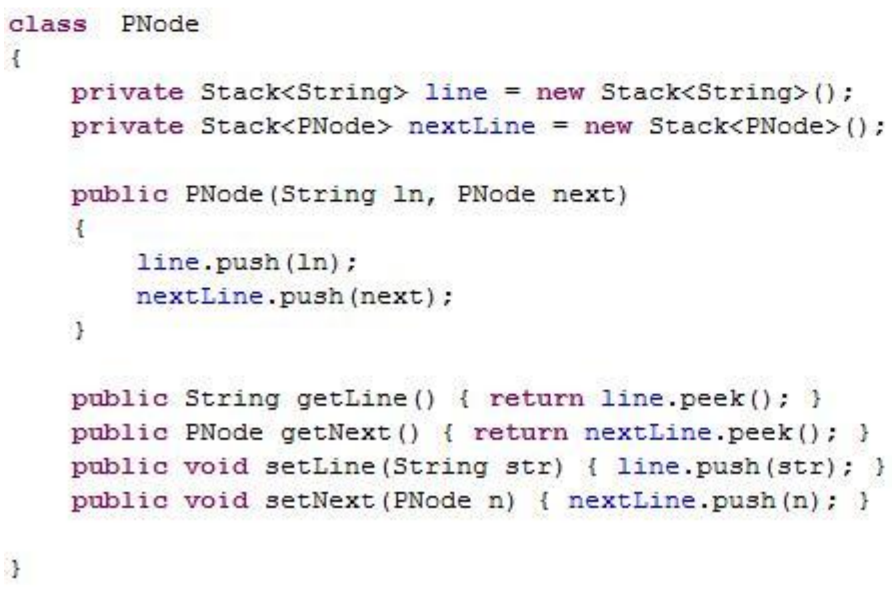
3.1.1 Making Nodes Persistent



```
class  /*persistent : PNode*/ Node{
      private String line;
      private Node nextLine;

      public Node(String ln, Node next){
                  line = ln;
                  nextLine = next;
      }

      public String getLine() { return line;  }
      public Node getNext() { return nextLine;  }
      public void setLine(String str) { line = str;}
      public void setNext(Node n) { nextLine = n; }
}
```

Figure 3-1 : Linked List Node

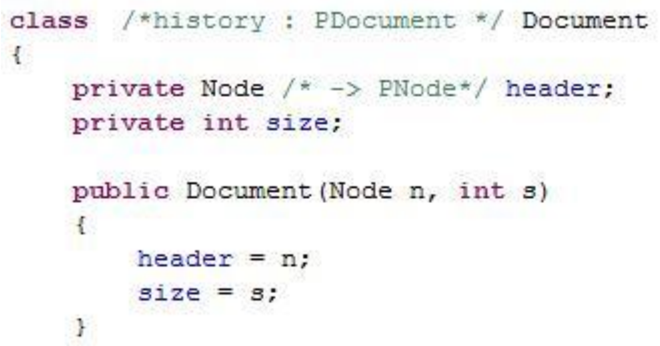
Figure 3-1 shows the code for the node class. The persistent annotation before the class name triggers the generation of a new class called `PNode`, which will be the persistent version of the `Node` class. There are two types of annotations for classes: persistent and history. When a class is marked with a `/*persistent : name*/` annotation, a new class with the specified name will be generated. In the new class, the fields of type `T` will be replaced with type `Stack<T>`, and the necessary rewritings will be performed in the class body. When a class is marked with a `/*history : name*/` annotation, in addition to the persistence changes, the generated class will have some additional fields such as a history stack.



```
class  PNode
{
    private Stack<String> line = new Stack<String>();
    private Stack<PNode> nextLine = new Stack<PNode>();

    public PNode(String ln, PNode next)
    {
        line.push(ln);
        nextLine.push(next);
    }

    public String getLine() { return line.peek(); }
    public PNode getNext() { return nextLine.peek(); }
    public void setLine(String str) { line.push(str); }
    public void setNext(PNode n) { nextLine.push(n); }

}
```

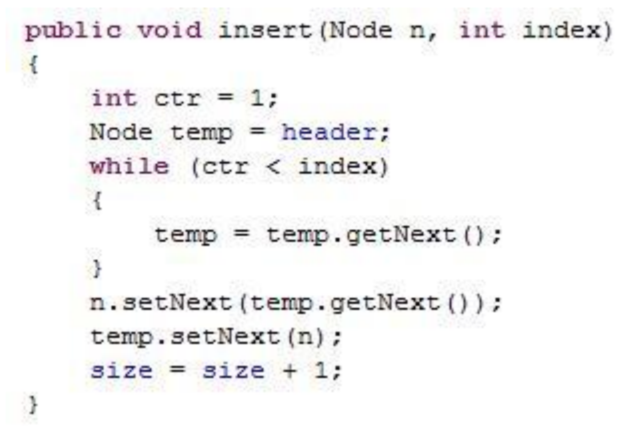
Figure 3-2: Persistent List Node

The generated code is shown in the above figure. In the class body all the accesses to the fields are replaced with `*field*.peek()`, and the assignments like `*field* = expr` are replaced with `*field*.push(expr)`.



```
class  /*history : PDocument */ Document
{
    private Node /* -> PNode*/ header;
    private int size;

    public Document(Node n, int s)
    {
        header = n;
        size = s;
    }
```

Figure 3-3: Document class and its constructor

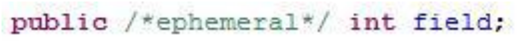


```
public void insert(Node n, int index)
{
    int ctr = 1;
    Node temp = header;
    while (ctr < index)
    {
        temp = temp.getNext();
    }
    n.setNext(temp.getNext());
    temp.setNext(n);
    size = size + 1;
}
```

Figure 3-4: Insert method of the Document Class

3.1.2 Making the Container Persistent

The `Document` class shown in Figure 3-3, is the implementation of a linked list class. It has two fields: a node, and an integer field that stores the size of the list. Its insert method, which inserts a given `Node` at a particular index, is shown in Figure 3-4. One restriction on persistent classes is that in order to avoid rewritings on the clients, their persistent fields have to be private. This way a persistent version of a class can be generated only by rewriting the code of the original class.

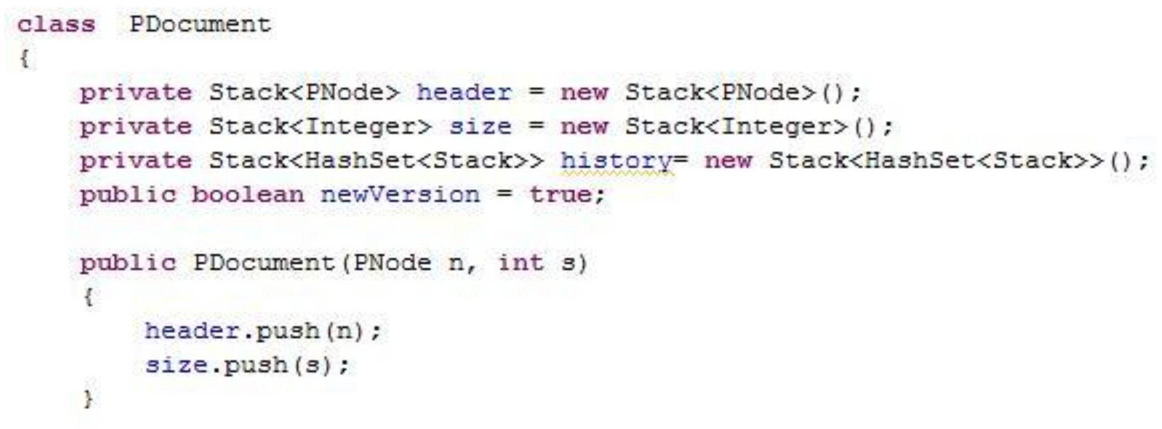


```
public /*ephemeral*/ int field;
```

Figure 3-5: An ephemeral field

Some fields can be marked with an `/*ephemeral*/` annotation as shown in Figure 3-5. Public ephemeral fields will remain as public in the generated code. Moreover, the types of those fields will not be rewritten in the persistent classes, and the changes on them will not be recorded in any history. In other words, they will not be affected by the undo method of the generated class, at all.

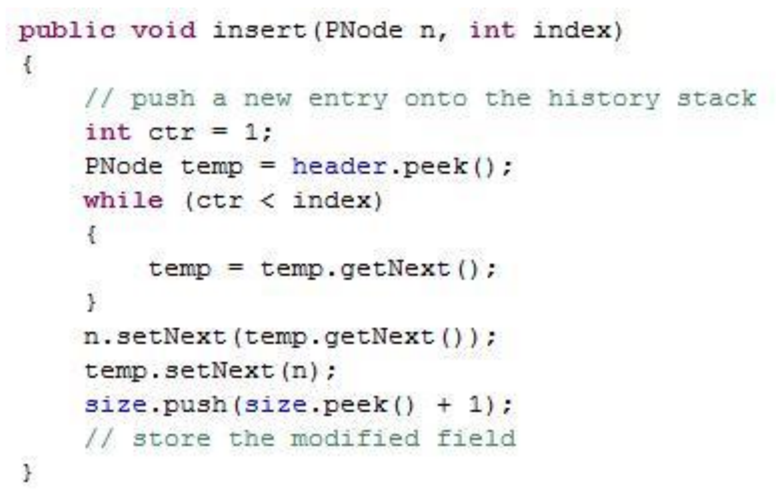
There is a new annotation `/* -> PNode*/` in Figure 3-3. It means that the type `Node` will be rewritten to `PNode` in the generated code. Other than fields of immutable and non-recursive types like `int`, `char`, `string` etc, all fields of an undoable structure must also be undoable. (It is very important that no undoable field is modified outside of its class. Even if it's private, a method can pass the field to a method of another class. In this case, if the field's own fields are not private, they can be modified within the other class. In that case the information will not be stored in the history. This case is being prevented because from transitivity, the fields of undoable fields themselves will be private. Therefore, they cannot be modified outside of their class.) The undoable version of type `Node`, which was named `PNode`, was generated before. Therefore it has to be used in the generated class. `Node` is rewritten to `Stack<PNode>`, and `int` is rewritten to `Stack<Integer>` (because Java does not allow parameterization by basic types).



```
class  PDocument
{
    private Stack<PNode> header = new Stack<PNode>();
    private Stack<Integer> size = new Stack<Integer>();
    private Stack<HashSet<Stack>> history= new Stack<HashSet<Stack>>();
    public boolean newVersion = true;

    public PDocument(PNode n, int s)
    {
        header.push(n);
        size.push(s);
    }
```

Figure 3-6: Persistent Document class and its constructor

Notice that there are two new fields in the `PDocument` class: `history` and `newVersion`. The former is the field that will store the modified references as mentioned earlier. It is a stack, and each of its elements will contain the modified references in an individual step. `newVersion` on the other hand, is there to let the programmers control what a step is. Its functionality will be explained later.



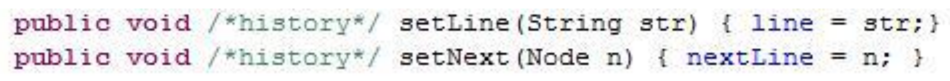
```
public void insert(PNode n, int index)
{
    // push a new entry onto the history stack
    int ctr = 1;
    PNode temp = header.peek();
    while (ctr < index)
    {
        temp = temp.getNext();
    }
    n.setNext(temp.getNext());
    temp.setNext(n);
    size.push(size.peek() + 1);
    // store the modified field
}
```

Figure 3-7: Sketch of the insert method of the persistent Document Class

Figure 3-7 shows the sketch of the generated insert function. What is missing in the figure is the code that saves the modified fields on the history stack which is indicated by comments. For the last line, it is trivial. The size field is incremented, so a reference to the size stack should be added to the top element of the history stack. However, note that the other modifications happen within the `setNext` method of the `PNode` class. We want to save the references to the modified stacks in the `PDocument` object's history.

3.2 Using `history`

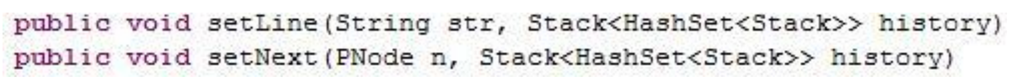
We introduce `/*history*/` annotations for methods in Figure 3-8, so that the methods can be parameterized in terms of history they will write to.



```
public void /*history*/ setLine(String str) { line = str;}
public void /*history*/ setNext(Node n) { nextLine = n; }
```

Figure 3-8: History annotations for methods

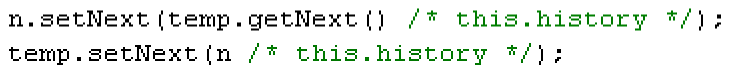
The signatures for the methods generated can be seen in Figure 3-9. The `setLine` method will store a reference to the `line` stack in the top element of the `history` parameter, and the `setNext` method will do the same for the `nextLine` stack.



```
public void setLine(String str, Stack<HashSet<Stack>> history)
public void setNext(PNode n, Stack<HashSet<Stack>> history)
```

Figure 3-9: Signatures of methods parameterized by `history`

Now to complete, we need to annotate the calls to the `setNext` method in the `insert` method of the `Document` class as shown in Figure 3-10. The corresponding method calls in the `PDocument` class will have two parameters, the actual ones in the code and `this.history`.



```
n.setNext(temp.getNext() /* this.history */);
temp.setNext(n /* this.history */);
```

Figure 3-10: Annotated method call

Note that one can use `null` as the history annotation. In that case, the modifications within the method will not be saved anywhere. For instance, one may not care about the temporary node being inserted in the list. So, if the annotation in the second line of Figure 3-10 is null, when the insert operation is undone, only the modified element of the list will be affected, not the temporary node.

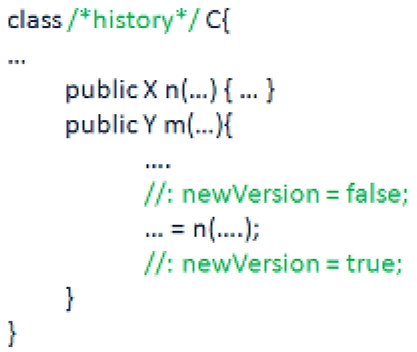
The insert method of the `PDocument` class also contains some code using the `newVersion` field, which was introduced in Figure 3-6. The field is used to determine whether the method should write to the current top element of the history stack, or should create a new one and write to that. The details of how to use the `newVersion` field will be explained in the next section.

Parameterized history gives the programmers the ability to choose where to store the modifications, and when to undo them. For instance, within the body of a method of an undoable class, a temporary instance of another undoable class can be created. The modifications on the temporary object can be saved in its own history, and they can be undone without interfering with the outer object.

3.3 Using `newVersion`

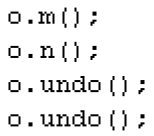
The modifications on the undoable structures are performed just like in [15]. Within an operation whenever an undoable field is updated, first the history is checked to see whether it was modified in this step before. If it was not, the new value is pushed onto the field's stack, and a reference to this stack is added to the top element of the history stack. If it was modified before, the top element of the field's stack is overwritten.

When we want to perform undo, the notion of a step is important. Consider Figure 3-11. The class C has public methods m and n. Notice that the method m contains a call to the method n. What should happen if n also modifies some of the fields m modifies? Most of the time, when m and n are called on an object, and two undos are performed like in Figure 3-12, we would expect to back to the initial state.



```
class /*history*/ C{
...
        public X n(...) { ... }
        public Y m(...){
                ....
                //: newVersion = false;
                ... = n(....);
                //: newVersion = true;
        }
}
```

Figure 3-11: Calling another method



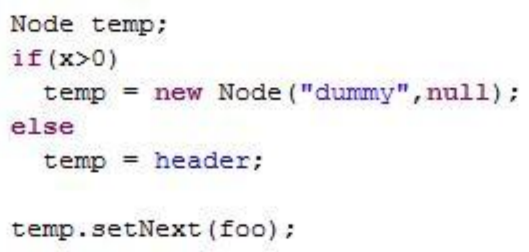
```
o.m();
o.n();
o.undo();
o.undo();
```

Figure 3-12: Undoing two method calls

This requires the two calls to the method `n` to behave differently. When `n` is called the first time in Figure 3-12, it is called in the body of method `m`. Undo simply works by traversing the top element of the history stack and popping the top of its elements. Therefore, to undo the whole execution of the method `m`, the modifications made by `m` and `n` should be recorded in the same history entry. When `n` is directly called on o, a new history entry will be created, and the modified fields will be recorded in it. However, when `n` is called within `m`, the modified fields will be recorded in the old entry. The field `newVersion` of undoable objects provides programmers a way to manipulate this kind of behavior and generalize the notion of a step. For each method operating on an undoable object, `newVersion` field of the object whose history the called method will use is important. (It will also be a passed as a parameter to the method along with history.) Whenever a method is called on an undoable object, first that `newVersion` field is checked. If it is true, and the top element of the history is not empty, a new history entry will be created and the modifications will be recorded in that. (Note that the last method performed on the object may be side effect free, and in that case an empty entry will remain in the history. The next time a method is called on this object, a new history entry will not be created, and the empty one will be reused.) Otherwise, the old entry will be used. The `newVersion` fields can be modified via annotations like `//: newVersion = true;` as shown in Figure 3-11. `//:` will be removed in the generated class.

3.4 Dealing with Aliasing

Driscoll, Sarnak, Sleator, and Tarjan[15] describe what to do in case fields of a persistent data structure is modified. However, in the presence of aliasing, it is not straightforward to identify what is being modified. For example, when something is modified through an alias, it may not be possible to know whether it is a local variable or a field of an undoable structure. Consider the code fragment in Figure 3-13. Depending on the value of `x`, `temp` may modify a local variable or the `Document` object.



```
Node temp;
if(x>0)
  temp = new Node("dummy",null);
else
  temp = header;

temp.setNext(foo);
```

Figure 3-13: Aliasing problem

The authors of [15] do not take aliasing into account; however, to implement their proposal, and generate persistent versions of ephemeral structure, one has to deal with aliasing. Ideally, whenever some part of a local variable is modified, it should not be added to the history. For instance, in Figure 3-7 when `temp` is modified within the loop, nothing will be saved. On the other hand, when a field of the `Document` object is modified through `temp` two lines below the loop, the modifications will be saved.

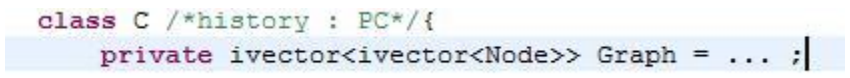
In general all potential aliases of a field of an undoable data structure will be computed, and all modifications through them will be added into the history conservatively. In some cases the modifications may be on local variables, and in that case undo will do some redundant work. One case to beware is, if the modification happens to be on a local variable, and there are no other modifications in that step. In that case, a redundant undo has to be performed on the object, to get rid of that entry in the history.

When the fields are modified through method calls as in Figure 3-13, the modifications on the parameters will be stored in the supplied history.

3.5 Limitations

The first restriction is that all of the undoable fields of an undoable data structure have to be private. This way no field can be manipulated outside the class, so there is no need to rewrite the clients of the undoable data structure. There is only one transformation on the ephemeral class which produces the persistent class.

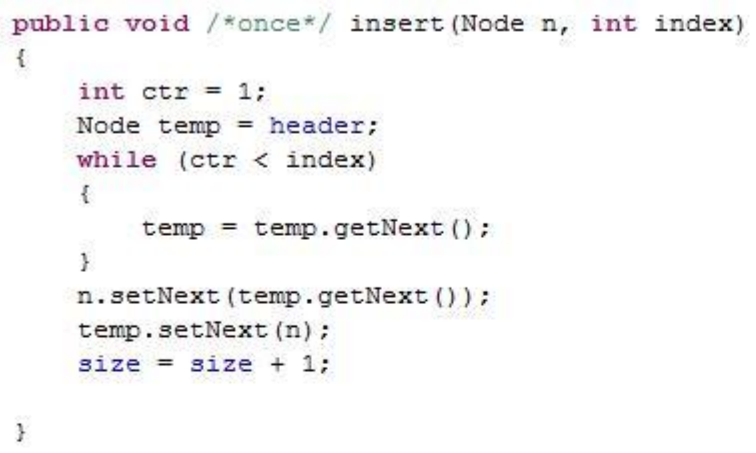
Another limitation is that data structures from existing libraries cannot be used within an undoable data structure in general. If they are used, the updates on them will not be undone. In order to undo the updates on a field, the type of the field has to be an undoable class. If the source codes of the libraries are available, they can be annotated and persistent versions of them can be generated. One particular case when classes without source codes can be used is when the class is immutable. Consider Figure 3-14. `ivector` is the immutable vector class. Here the Node class is mutable. Whenever a Node of the field Graph is modified in the undoable class PC, if the history of the PC object is passed to the methods of the Node class, all changes on the PC object can be undone.



```
class C /*history : PC*/{
    private ivector<ivector<Node>> Graph = ... ;
```

Figure 3-14: Using immutable classes

There is a performance limitation. Every time a field is modified, it is looked up in the top element of the history to see it was modified within the step before. This operation is expensive. Many methods, like the insert method of the Document class, and the setNext method of the Node class modify fields at most once. We introduce the /*once*/ annotation (see Figure 3-15) for these kind of methods, to make use of this observation, and avoid the look up.

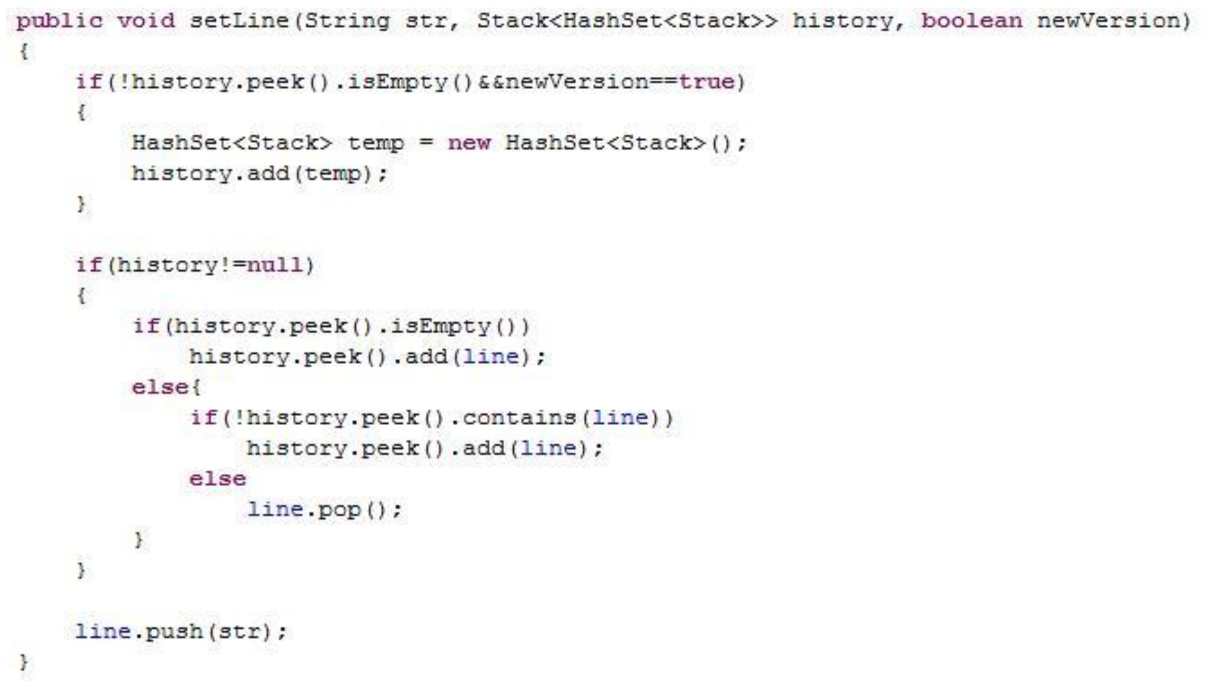


```
public void /*once*/ insert(Node n, int index)
{
    int ctr = 1;
    Node temp = header;
    while (ctr < index)
    {
        temp = temp.getNext();
    }
    n.setNext(temp.getNext());
    temp.setNext(n);
    size = size + 1;

}
```

Figure 3-15: Once annotation for efficiency

Note that even if a method modifies the fields at most once, those fields may be modified by methods called before it within the same step. Therefore, we cannot blindly eliminate the look ups in the methods marked with the /*once*/ annotation. However, we can use runtime information. If the history is empty when the method starts executing, it means that no fields were modified before. Therefore, we can eliminate the look ups while storing the modified fields in that case. Note that this is a more general way than checking if the `newVersion` field is true or not. Even if the `newVersion` is set to false to use the same history entry in the subsequent method call, the history can still be empty.

Another way of eliminating look-ups is simply to check whether the history is empty before the modification. If the history is empty, there is no need to do the look-up.



```
public void setLine(String str, Stack<HashSet<Stack>> history, boolean newVersion)
{
    if(!history.peek().isEmpty()&&newVersion==true)
    {
        HashSet<Stack> temp = new HashSet<Stack>();
        history.add(temp);
    }

    if(history!=null)
    {
        if(history.peek().isEmpty())
            history.peek().add(line);
        else{
            if(!history.peek().contains(line))
                history.peek().add(line);
            else
                line.pop();
        }
    }

    line.push(str);
}
```

Figure 3-16: Transformed setLine method

As an example to a fully transformed method, Figure 3-16 shows the transformed version of the `setLine` method from Figure 3-1. The original method body is just a single line and assigns the parameter `str` to the field `line`. The last line in the transformed method corresponds to it. The two preceding if blocks are created by the transformation. The first if block is common to all methods writing to a history. It checks if the last entry in the history is not empty and the newVersion parameter passed is equal to true. If so, a new entry will be added to the history, and the modifications will be recorded in that. Otherwise the current entry in the history stack will be used. The second if block is added for the original assignment statement. As explained earlier, it first checks whether the last element is empty. If it is, it records the information that `line` will be modified, without performing any look-ups on the existence of a reference to `line` in the top element of the history stack.

Otherwise, it performs the look-up for `line`, and records the same information if a reference to `line` is not present in the top element of the history stack. If it is present, it pops the top of the `line` stack, so that only one entry per version exists, i.e. the new value of `line` is rewritten instead of the old one like overwriting a field.

3.6 Implementation Notes

The input programs of the tool are in a valid subset of Java. All the annotations are added as special comments, so the source (ephemeral) classes can be used as they are. The generated code is also valid Java code, and can be executed with a Java compiler. The grammar of the input language is specified in Appendix A. It can be characterized as Featherweight Java + arrays + assignment + import statements. It is not an exact definition, but a close enough one. The grammar only specifies the language, and does not include the annotations. All of the annotations are explained in previous sections of this chapter. The implementation language of the tool is F#.

Chapter 4

Applications

In this chapter, we focus on specific applications of undoable objects; namely filtering context-free grammar objects, and building application level undo functionality for interactive applications.

4.1 Constraint Programming

Our particular choice of paradigm to investigate backtrack operation on is Constraint Programming due to reasons such as its expressiveness, wide range of applicability and more importantly its inherent need for a non-trivial backtrack operation compared to, for instance, Boolean Satisfiability (SAT) where literals can only be assigned to *true* or *false* hence flipping values is enough for backtracking. We first briefly explain the computational model in constraint programming and provide an intuition into how undoable objects would make the programming effort easier compared to the traditional methods.

We first introduce some basic constraint programming concepts. For more information we refer to [6]. A *constraint satisfaction problem* (CSP) P is a triple P = <X,D,C> where X is a n-tuple of variables $X =< x_1, . . . , x_n >$, D is corresponding domains of variables $D =< d_1, . . . , d_n >$ such that $x_i \in di, \forall i \in 1 . . . n$, and C is a set of constraints $C = \{c1, . . . , ck\}$ where each $c_k \in C$ specifies allowed combinations of values to a subset of the variables in X. Constraints of arity 2 are called binary constraints and constraints defined by an arbitrary arity are called global constraints. A constraint can be specified extensionally by a list of its satisfying tuples, although, it is intractable to enumerate all satisfiable tuples in practice. Global constraints are used to express ubiquitous patterns (e.g. a set of variables must take different values, notice that the size of the pattern is not fixed). It is important to incorporate global constraints in constraint solvers so that users can express such common patterns easily. However it is also important to have a way to propagate them without using generic arc consistency algorithms since optimal generic arc consistency algorithms require $O(rd^r)$ time for constraints involving r variables with maximum domain size d. Furthermore, these filtering algorithms must be accompanied by customized backtracking operators.

An assignment gives each variable a value from its domain and a solution is an assignment which satisfies all constraints. Broadly, constraint solvers apply two techniques to solve CSPs; inference and search and their various combinations. Inference is conducted via *constraint propagation* while a systematic search is performed to explore the finite search space. The search process constructs a search tree by enumerating all possible variable-value combinations until we find a solution or prove that none exists while constraint propagation is applied at each node to reduce the exponential number of combinations. Constraint propagation removes domain values that do not belong to a solution with respect to a constraint. The removal of inconsistent values is called *filtering* and the process is repeated for all constraints until no more domain values can be removed. The relevance of this computational model to this thesis is that whenever a value is removed from a domain, the state changes and we move to a next state whereby a backtrack operation is needed to undo the changes that we have committed. In order to be effective, filtering algorithms should be efficient, because they are applied many times during the solution process. They should also be efficient as they accompany the filtering during search.

One of the major directions of research in CP is to identify common combinatorial structures in discrete constraint problems and to provide highly *expressive* global constraints. Currently, CP provides more than two hundreds of global constraints. Without an exception all of these constraints use the benefits of incremental filtering algorithms which work on non-trivial data structures. Hence, how to rollback the changes that we committed on these data structures throughout the search is an important question. ILOG [17], a leading CP

Solver, utilizes *Reversible Integers* for this purpose. Reversible integers are integer objects which also keep a copy of their previous states together with their current value. Whenever a backtrack operation is called, it is trivial to undo changes on these objects since we have access to all previous states. This approach works fine as long as the only structure needed for a filtering algorithm is integers (for example; flow constraints). However, in a case where using only integers is not enough to express a filtering algorithm for a constraint, that is custom data structures are needed, the user has to manually implement a backtrack operation as well as implementing the filtering algorithm itself.

We consider such a global constraint; Context-Free Global Constraint [20]. This constraint represents variables as a string and enforces it to be derived from a given grammar. Using reversible integers is not enough to express the underlying filtering algorithm hence a customized data structure and a manual backtracking operator is required. We demonstrate the filtering algorithm for context-free grammar constraints and the corresponding non-trivial backtracking operation and propose to use undoable objects in order to generate backtracking function automatically.

4.1.1 Context-Free Grammar Constraint Filtering

The incremental filtering algorithm for this constraint is based on Cooke, Younger, Kasami (CYK) parsing algorithm. The filtering works in two phases. First, we work bottom-up by computing the sets of non-terminals that can produce subsequences of the desired strings. If the last created non-terminal is the starting non-terminal, when the whole length is considered, the CSP is satisfiable, otherwise there is no satisfying assignment. In the former case, the algorithm works top-down by removing all non-terminals which cannot be reached from the start symbol and consequently performs domain filtering.

Let us consider the following example.

Example: Assume we are given the following context-free, normal-form grammar G = ({ [,] } {A, B, C, S0}, { S0 → AC, S0 → S0 S0, S0 → BC, B → A S0, A→ [, C→] }, S0) that gives the language LG of all correctly bracketed expressions(like, for example "[[][]]" or "[][[]]").

We illustrate how the filtering algorithm works when the initial domain of all variables is { [,] }. First, as seen in Figure 4-1, we work bottom-up and add non-terminals if they allow us to generate a word of a certain length. For instance, the non-terminals C and A is placed in the row because every variable has a of { [,] } and C and A are the production rules that produces these letters. Similarly, the non-terminal S_0 in the cell (1,2) is created since we have a production rule S0 → AC , and A from cell (1,1) and C from (2,1) give their support to produce a word of length two.

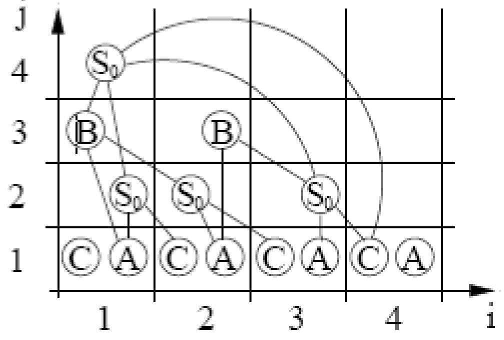


Figure 4-1: The algorithm first works bottom-up creating sets of non-terminals

In Figure 4-2, the algorithm works top-down and removes all non-terminals that cannot be reached from S0.

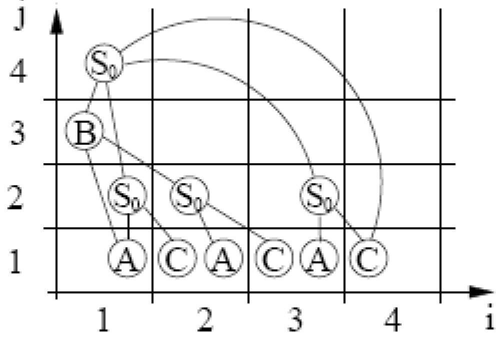


Figure 4-2: The algorithm works top-down removing non-terminals that are unreachable from the starting symbol.

Notice that in Figure 4-2, left and right parenthesis are removed from the domain of the first and the last variable, since we know any correctly bracketed sentence cannot start with a closing parenthesis and end with an opening parenthesis.

Now we assume that a branching decision has been made and left parenthesis (labeled as #1 in the figure below) is removed from the domain of the third variable. This filtering event will have an immediate effect on the data structure since some non-terminals will lose their support. Specifically; nodes labeled as 1, 2 and 3, as numbered in the order of removal.

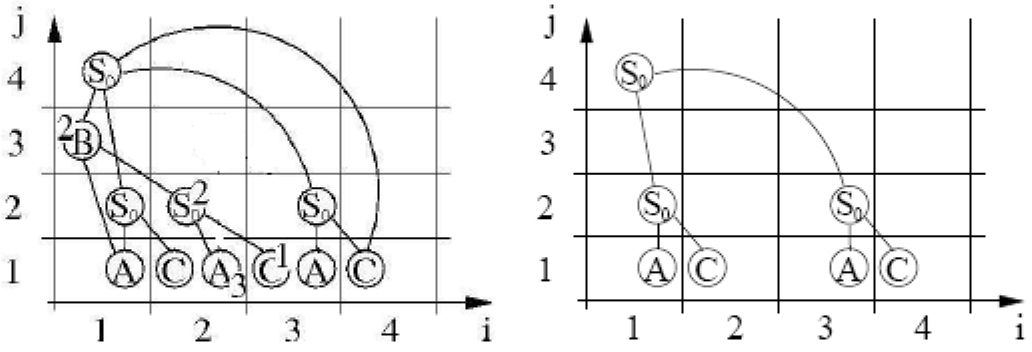


Figure 4-3

In this incremental filtering algorithm, the data structure is not created from scratch for every branching decision but only a subpart of the original structures is changed. Therefore, we need a way to undo the changes that we have committed and to restore back to a previous state. That is, once the left parenthesis made available for the third variable, we need a way to go back from Figure 4-3 to Figure 4-2. Such an explicit backtrack operation has to be implemented manually and given such non-trivial data structures, this work is very error prone. We argue that if the data structure used above is implemented through undoable objects we can roll back to a previous state of the graph automatically within our method without requiring any explicit implementation. One future direction of our work is to assess the efficiency of our approach by comparing the manually written backtrack operator with our automatically generated version.

4.2 Undo for Interactive Applications

Undo is crucial in interactive applications such as editors, and program development environments for users to recover from their errors. Users frequently make mistakes, or change their minds. Therefore interactive applications have to provide facilities to reverse the effects of past operations, and to restore the state of objects.

We give a simple text editor example to demonstrate how undoable objects make building larger application level undo functionality easier. Consider a text editor that uses a persistent document class. The operations that the user can perform on the document will each correspond to a step. (Typically all of them would be separate methods of the document class that possibly call internal methods.) Note that for the persistent class, undo for all the external operations comes for free.

Multiple documents can be edited at the same time. In that case, there are multiple alternatives for the semantics of the applications undo. For example, one alternative is that the application undo will undo the changes of the document in the active window. Another alternative would be sequencing all the changes on the documents, and undoing the last one regardless of the active view. The first undo is very straightforward to implement. In that case, the application level undo method will only need to call the undo method of the active document which was generated automatically. The second undo requires a little more work. Assume that all the documents are stored in an array. Each time a document is modified, its index within the array will be pushed onto a log stack. Then the application level undo will read the top of the stack, and call the undo method of the indexed document object.

An alternative way of implementing the text editor is having a class with a persistent arraylist field. The arraylist will hold the individual documents. Just like in the linked list example, the changes done on the documents will be saved in the history of the outer class. Note that in this case, we do not need any glue clode (like the additional undo stack) to implement undo. However, this only allows implementing undo with the second semantics mentioned previously.

Notice that when the editor is extended with additional operations that manipulate documents, undo for all the operations will be inherited for free.

Chapter 5

Related Work

Considerable amount of persistent data structures [9, 14, 22, 27] have been developed over the years. The early motivation for persistent data structures came from computational geometry [29]. Apart from the specific data structures, Driscoll, Sarnak, Sleator, and Tarjan [15] developed the first efficient systematic ways of making a data structure persistent as an outgrowth of Sarnak's Ph.D. thesis [28]. In his survey paper Kaplan [21] gives a comprehensive account of the research on persistence.

Conchon and Filliâtre [11] introduced *semi-persistent* data structures especially for backtracking algorithms. When a persistent branch ends within a backtracking algorithm, there is no need to undo the modifications performed on the data structures. A new branch can be started with the old version. In this setting, full persistence is not required since only the ancestors of the current version are used, but never the siblings. A data structure is called semi persistent if only the ancestors of the newest version can be updated. It is different than partial persistence since the ancestors can be updated, not only accessed. Conchon and Filliâtre also proposed a proof system to statically check the legal use of semi persistent data structures based on user annotations.

The notion of purely functional data structure is closely related to persistence. A purely functional data structure is a data structure that does not contain any assignment statements. Note that, such a structure is automatically persistent; however, the converse is not true. Imperative data structures can be persistent, too. Okasaki's book [26] gives a good overview of purely functional data structures.

Another way of implementing undo on mutable objects is via first class stores [18, 25]. Duggan and Johson describe the language GL. Its features include partial continuations, and first class stores. They also use the persistence methods in [18] to implement first class stores. Morrisett [25] generalized the notion and showed how to partition the mutable data and refine the scope of store objects. First class store is an advanced language feature. We took an alternative approach and stayed within Java. Our model does not require any additional language feature.

Leeman [24] was the first to study how to add undo into the programming process. He proposed two formal models of linear undo without redo. The paper also includes a very good overview of the applications of undo.

Another related topic is reverse execution of programs. One important application of reverse execution is debuggers such as [1, 2, 33]. There are three general approaches for reverse execution: *logging*, *checkpointing*, and *reverse code generation*.

Logging is to store the data necessary for reverse execution as the program runs forward. The Java bytecode debugger of [12] that can run the code backwards uses logging. JVM is a stack based virtual machine. Almost all of its instructions can be characterized as popping m items from the operand stack, possibly operating on them, and pushing n items onto the operand stack. A circular log simply records the popped m items before the instruction is executed, and the n items are discarded when the instruction is reversed. In case of branches and jumps, information on the control flow is also stored to determine the order in which the instructions are executed. The log is circular, so that when the allocated space runs out, front end will be overwritten.

Logging is not a memory efficient approach. In checkpointing not all the state changes are saved, but periodic checkpoints are gathered. To go back to a particular point, the closest check point is restored, and the program is re-executed until the desired point. Tolmach and Appel's Standard ML debugger [33, 34] and [1] are based on checkpointing. In Tolmach and Appel's work, checkpoints are continuations. (Mutable state is not stored in the continuation, and they handle it differently.)

Reverse code generation is the most memory efficient on among the three approaches. In this approach, state saving is used as a last resort, and reverse execution code is generated for every reversible statement. Notable examples of the reverse code generation techniques include [2, 3, 4, 16, 23]. Akgul and Mooney's work [2, 3, 4] is the best one in terms of memory efficiency. In some cases they can even compute the reverses of destructive updates like `x = 5;` without saving any state. They work in the assembly code level and the granularity of reversal is individual instructions. The best part of their work is almost everything is done statically without any runtime overhead. They work on the control flow graph of procedures. For each procedure a reverse procedure is calculated statically. The CFG doesn't give direct information about the dynamic control flow. For example, some basic blocks can be reached from different blocks depending on certain predicates. The dynamic control flow information is recovered by computing control flow predicates. In a later paper [5] they enhanced their technique with dynamic slicing in order to reverse just the statements related to the particular bugs being examined.

Chapter 6

Conclusion

Based on the notion of persistence we introduced undoable objects. Persistent versions of classes are generated automatically from the ephemeral classes, and undo is taken for free. The automatic generation of the persistent classes is guided by user annotations, whose overhead is much less compared to implementing undo from scratch. After the generation of persistent version of a class, it can be used just like the ephemeral version. As both the input (ephemeral classes) and the output (persistent classes) are plain Java code, we do not need any extra language feature.

6.1 Future Work

The programming power and the expressivity of the model presented in the thesis can be enhanced in a number of ways. The current undo model is a linear one – effects of a method `m` called on an object cannot be undone before undoing the effects of all methods that were called on the object after `m` – and one important question is how to incorporate selective undo into this model. Linear undo is suitable for chronological backtracking, however it is not useful in implementing algorithms based on non-chronological backtracking [13] algorithms. Selective undo may help in this regard.

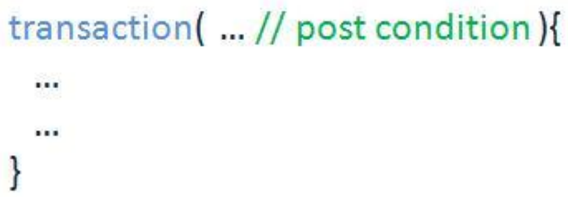
`newVersion` and `history` fields generalize the notion of a step and provide means to construct arbitrary undo operations. However, the solution offered by the `newVersion` field is an ad-hoc one and creates a state dependency. Considering the fact that most of the time we do not need that generality (for example the problem created by a public method calling another public method of the class occurs very rare), a study of dynamic execution contexts may provide a cleaner, and an object oriented solution.

Another useful edition would be automatically generating redo methods for objects. If we use trees instead of stacks as fat nodes, then we can reach older versions of a field with logarithmic slowdown. In order to support redo, we will not pop the top of the stack when undo is performed, but only record information about the most recent version. If redo is called on an object, then only the most recent version information will be updated. Instead of redo, if another method is called after a number of undos, all obsolete versions will be removed. Note that with this implementation strategy, there is a logarithmic slowdown for reading values of undone fields after performing any number of undos.

Automatically generating a commit method may be useful for some applications. After a while the history may become unnecessary and the older versions can be removed via commit.

There is still much to be done to fully evaluate the viability of the approach taken in this thesis. As noted in Chapter 4, a good evaluation would be rewriting the Context Free Grammar Constraint [19, 20] using persistence and comparing its performance with the manually written backtracking on real life scheduling problems.

The `/*once*/` annotations require trust in programmers. The programmer may incorrectly mark a method which modifies some field more than once, and in that case undo will not work correctly. Instead of using trusted annotations, static analysis techniques can be used to find out such methods in a sound way. One drawback of this approach would be its considerable annotation overhead. In order to do any useful analysis, the programmers would need to annotate methods with pre and post conditions, and control aliasing (for example, if the method takes two parameters of the same type and modifies both parameters exactly once, we need to make sure that these two parameters will never be aliases of each other).



```
transaction( ... // post condition ){
   ...
   ...
}
```

Figure 6-1: Transaction construct

Based on undoable objects, Java can be extended with a transaction construct as shown in Figure 7-1. The transaction construct takes a boolean expression as a post-condition, and a code block to execute. If the code block throws an exception, or the post-condition evaluates to false, every modification on persistent structures within the code block will be undone. Otherwise, the modifications will persist. As they do not have undo methods, ephemeral objects should not be modified in a transactional block.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to program debugging. *IEEE Software* May 1991.

[2] T. Akgul. *Assembly Instruction Level Reverse Execution for Debugging*. PhD. Thesis, Georgia Institute of Technology, Spring 2004.

[3] T. Akgul and V. J. Mooney III. Instruction-level reverse execution for debugging. *Workshop on Program Analysis For Software Tools and Engineering*, 2002.

[4] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transactions on Software Engineering and Methodology*, 13(2):149–198, April 2004.

[5] T. Akgul, V. J. Mooney III, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. *26th International Conference on Software Engineering* (ICSE '04), 2004.

[6] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[7] J. E. J. Archer, R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, Jan. 1984.

[8] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

[9] G. S. Brodal and C. Okasaki. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, Volume 6(6), 1996.

[10] A. B. Brown. *A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors With System-Wide Undo*. PhD. Thesis, UC Berkeley, Dec. 2003.

[11] S. Conchon and J.C. Filliâtre. Semi-Persistent Data Structures. *17th European Symposium on Programming*, 2008.

[12] J. J. Cook. Reverse Execution of Java Bytecode. *The Computer Journal*, 2002.

[13] R. Dechter and D. Frost. Backjump-based Backtracking for Constraint Satisfaction Problems, *Artificial Intelligence*, 2002.

[14] P.F. Dietz. Fully persistent arrays. *Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, Vol 382. Springer-Verlag, ACM, New York, 1989.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *18th ACM Symposium on Theory of Computing*, May 1986.

[16] D. Gries. *The Science of Programming*. Springer-Verlag New York, 1987.

[17] ILOG Solver. http://www.ilog.com . August 13, 2009.

[18] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and environment. *15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1988.

[19] S. Kadioglu. *Symbollic Constraints: Combining Expressiveness with Efficiency*. Master Thesis, Brown University, 2009.

[20] S. Kadioglu and M. Sellmann. Efficient Context-Free Grammar Constraints. *AAAI*, AAAI Press, 2008.

[21] H. Kaplan. Persistent Data Structures. *Handbook of Data Structures and Applications*. CRC Press, 2005.

[22] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 31:11-16, 1999.

[23] J. Lee. Dynamic Reverse Code Generation for Backward Execution. *Electronic Notes in Theoretical Computer Science*, 174(4):37-54, 2007.

[24] G. B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50-87, Jan. 1986.

[25] J. G. Morrisett. Refining First Class Stores. *ACM SIGPLAN Workshop on State in Programming Languages*, 1993.

[26] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[27] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors, ACM Trans. on Programming Systems and Languages. 5(193), 449-477.

[28] N. Sarnak. *Persistent Data Structures*, PhD. Thesis, Dept. New York University, New York, 1986.

[29] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees, *Communications of the ACM 29*, 1986.

[30] R. M. Stallman. *GNU Emacs Manual*, 15th Ed. Boston, MA: GNU Press, 2002.

[31] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563-573, Sept. 1981.

[32] W. Teitelman. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, 1978.

[33] A. P. Tolmach. *Debugging Standard ML*. PhD. Thesis, Princeton University, Oct. 1992.

[34] A. P. Tolmach and A.W. Appel. Debugging Standard ML without reverse engineering. *ACM Conference on Lisp and Functional Programming*, June 1990.

Appendix A

Grammar of the Language

Program:
    ImportDeclarationsOpt ClassDeclarationsOpt EOF

Literal:
    INT_LITERAL
| STR_LITERAL
| CHAR_LITERAL
| NULL

ArrayType:
    PrimitiveType LB RB
| *Name* LB RB
| *ArrayType* LB RB

Name:
    IDENT
| *Name* DOT IDENT

ImportDeclarations:
    ImportDeclaration
| *ImportDeclarations ImportDeclaration*

ImportDeclarationsOpt:
    /* empty */
| *ImportDeclarations*

ImportDeclaration:
    SingleTypeImportDeclaration
| *TypeImportOnDemandDeclaration*

SingleTypeImportDeclaration:
    IMPORT *Name* SM

TypeImportOnDemandDeclaration:
	IMPORT *Name* DOT TIMES SM

Type:
	PrimitiveType
|	*ReferenceType*

PrimitiveType:
	INT
|	CHAR
|	BOOL

ReferenceType:
	ClassType
|	*ArrayType*

ClassType:
	Name

ClassDeclarations:
	ClassDeclaration
|	*ClassDeclarations ClassDeclaration*

ClassDeclarationsOpt:
	/* empty */
|	*ClassDeclarations*

ModifierOpt:
	/* empty */
|	*Modifier*

Modifier:
	PUBLIC
|	PRIVATE

ClassDeclaration:
	CLASS IDENT EXTENDS *Name ClassBody*

ClassBody:
	LC *ClassBodyDeclarationsOpt* RC

ClassBodyDeclarations:
	ClassBodyDeclaration
|	*ClassBodyDeclarations ClassBodyDeclaration*

ClassBodyDeclarationsOpt:
/* empty */
| *ClassBodyDeclarations*

ClassBodyDeclaration:
ClassMemberDeclaration
| *ConstructorDeclaration*

ConstructorDeclaration:
ModifierOpt ConstructorDeclarator ConstructorBody

ConstructorDeclarator:
IDENT LP *FormalParameterListOpt* RP

ConstructorBody:
LC *BlockStatementsOpt* RC

ClassMemberDeclaration:
FieldDeclaration
| *MethodDeclaration*

FieldDeclaration:
ModifierOpt Type VariableDeclarator SM

MethodDeclaration:
MethodHeader MethodBody

MethodBody:
Block
| SM

MethodHeader:
ModifierOpt Type MethodDeclarator

MethodDeclarator:
IDENT LP *FormalParameterListOpt* RP

FormalParameterList:
FormalParameter
| *FormalParameterList* CM *FormalParameter*

FormalParameterListOpt:
/* empty */
| *FormalParameterList*

FormalParameter:
    Type VariableDeclaratorId

ExpressionParameterList:
    Expression
| *ExpressionParameterList* CM *Expression*

ExpressionParameterListOpt:
    /* empty */
| *ExpressionParameterList*

Block:
    LC *BlockStatementsOpt* RC

BlockStatementsOpt:
    /* empty */
| *BlockStatements*

BlockStatements:
    BlockStatement |
    BlockStatements BlockStatement

BlockStatement:
    LocalVariableDeclaration
| *Statement*

LocalVariableDeclaration:
    Type VariableDeclarator SM

VariableDeclarator:
    VariableDeclaratorId
| *VariableDeclaratorId* ASSIGN *VariableInitializer*

VariableDeclaratorId:
    IDENT
| *VariableDeclaratorId* LB RB

VariableInitializers:
    VariableInitializer
| *VariableInitializers* CM *VariableInitializer*

VariableInitializer:
    Expression
| *ArrayInitializer*

ArrayInitializer:
        LC *CommaOpt* RC
| LC *VariableInitializers CommaOpt* RC

CommaOpt:
        /* empty */
| CM

Statement:
        StatementWithoutTrailingSubstatement
| *IfThenStatement*
| *IfThenElseStatement*
| *WhileStatement*

StatementNoShortIf:
        StatementWithoutTrailingSubstatement
| *IfThenElseStatementNoShortIf*
| *WhileStatementNoShortIf*

StatementWithoutTrailingSubstatement:
        Block
| *EmptyStatement*
| *ExpressionStatement*
| *ReturnStatement*

EmptyStatement:
        SM

ExpressionStatement:
        StatementExpression SM

StatementExpression:
        Assignment
| *MethodInvocation*
| *ClassInstanceCreation*

IfThenStatement:
        IF LP *Expression* RP *Statement*

IfThenElseStatement:
        IF LP *Expression* RP *StatementNoShortIf* ELSE *Statement*

IfThenElseStatementNoShortIf:
        IF LP *Expression* RP *StatementNoShortIf* ELSE *StatementNoShortIf*

WhileStatement:
WHILE LP *Expression* RP *Statement*

WhileStatementNoShortIf:
WHILE LP *Expression* RP *StatementNoShortIf*

ReturnStatement:
RETURN *ExpressionOpt* SM

Primary:
PrimaryNoNewArray
| *ArrayCreationExpression*

PrimaryNoNewArray:
Literal
| THIS
| LP *Expression* RP
| *ClassInstanceCreation*
| *FieldAccess*
| *MethodInvocation*
| *ArrayAccess*

ClassInstanceCreation:
NEW Name LP *ExpressionParameterListOpt* RP

ArrayCreationExpression:
NEW *PrimitiveType DimExprs DimsOpt*
| NEW *PrimitiveType Dims ArrayInitializer*
| NEW *Name DimExprs DimsOpt*
| NEW *Name Dims ArrayInitializer*

DimExprs:
DimExpr
| *DimExprs DimExpr*

DimExpr:
LB *Expression* RB

Dims:
LB RB
| *Dims* LB RB

DimsOpt:
/* empty */
| *Dims*

FieldAccess:
Primary DOT IDENT

MethodInvocation:
Name LP *ExpressionParameterListOpt* RP
| *Primary* DOT IDENT LP *ExpressionParameterListOpt* RP

ArrayAccess:
Name LB *Expression* RB
| *PrimaryNoNewArray* LB *Expression* RB

PostfixExpression:
Primary
| *Name*

UnaryExpression:
PLUS *UnaryExpression* |
MINUS *UnaryExpression*
| *UnaryExpressionNotPlusMinus*

UnaryExpressionNotPlusMinus:
PostfixExpression
| NOT *UnaryExpression*
| *CastExpression*

CastExpression:
LP *PrimitiveType* RP *UnaryExpression*
| LP *Expression* RP *UnaryExpressionNotPlusMinus*
| LP *ArrayType* RP *UnaryExpressionNotPlusMinus*

MultiplicativeExpression:
UnaryExpression
| *MultiplicativeExpression* TIMES *UnaryExpression*
| *MultiplicativeExpression* DIV *UnaryExpression*
| *MultiplicativeExpression* MOD *UnaryExpression*

AdditiveExpression:
MultiplicativeExpression
| *AdditiveExpression* PLUS *MultiplicativeExpression*
| *AdditiveExpression* MINUS *MultiplicativeExpression*

```
RelationalExpression:
        AdditiveExpression
|       RelationalExpression LT AdditiveExpression
|       RelationalExpression GT AdditiveExpression
|       RelationalExpression LE AdditiveExpression
|       RelationalExpression GE AdditiveExpression

EqualityExpression:
        RelationalExpression
|       EqualityExpression EQ RelationalExpression
|       EqualityExpression NOT_EQ RelationalExpression

AndExpression:
        EqualityExpression
|       AndExpression AND EqualityExpression

OrExpression:
        AndExpression
|       OrExpression OR AndExpression

AssignmentExpression:
        OrExpression
|       Assignment

Assignment:
        LeftHandSide ASSIGN  AssignmentExpression

LeftHandSide:
        Name
|       FieldAccess
|       ArrayAccess

ExpressionOpt:
        /* empty */
|       Expression

Expression:
          AssignmentExpression
```