Implementing Sets for Reversible Computation

Bill Stoddart and Frank Zeyda
School of Computing and Mathematics
University of Teesside, North Yorkshire, U.K.

August 17, 2002

Abstract

Sets provide a very general tool for representing information and modelling the behaviour of systems. We consider their implementation and associated problems of garbage collection in the context of reversible computation. We describe our implementation technique, which uses ordered arrays, and discuss scalability of performance.

1 Introduction

Sets provide a very general tool for representing information and modelling the behaviour of systems. We can use well defined mathematical operations to construct sets and extract information from them, and can call on the mathematics of set theory to reason about such systems. On the other hand, the implementation of sets cannot directly take advantage of the sequential nature of computer memory, and can result in the generation of garbage. Thus a programming language which uses sets is generally more easy to analyse but less efficient to implement that a language whose data structures follow patterns suggested by the sequential physical organisation of computer memory.

We consider these issues in the context of a reversible Forth which is being developed to act as an intermediate language for a compiler. The high level source language for the compiler is based on GSL, Jean-Raymond Abrial's "Generalised Substitution Language", which is the basis of the B Method, a technique for developing high integrity software supported by mathematical proof [1] and based on set theory [3].

Reversibility brings backtracking and other techniques for constraint resolution [5], [4] and is easily incorporated within the mathematical analysis used by the method.

We remind the reader that sets are collections of elements, in which order and repetition have no significance $\{1, 2, 3\}$ represents the set containing the numbers 1, 2 and 3. Exactly the same set can be written as $\{3, 2, 1\}$ or $\{1, 2, 2, 3\}$ and in many other ways.

In this paper we consider the implementation of sets of primitive objects (numbers and strings) and also structured sets, which may contain ordered pairs of objects and sets of objects in arbitrary combinations. Ordered pairs, written (x, y), allow us to model associations between data (relations). For example here is part of the relation between authors and articles for 16th EuroForth.

{ (“S Pelc”, { “The VFX optimiser”, “DOCGEN” }),
(“A Ertl”, {“CONST-DOES”}), ... }

The relation is given between authors and sets of articles, e.g. S Pelc has written two articles, ”The VFX Optimiser” and ”DOCGEN”.

Sets can be used to model and manipulate data of arbitrary complexity and the formal description of operations performed on this data will usually be simpler than if any other representation is used. On the other hand execution efficiency may be compromised.

2 Set literals, set operations and garbage

Consider for a moment what happens during the interaction:

```
1 2 + .  3 ok
```

The literal values 1 and 2 are created on the stack and consumed by the addition, which leaves no trace of their existence. Now let us consider what is different in the following set calculation:

```
{ 1 , 2 } { 2 , 3 } UNION .SET  { 1 , 2 , 3 } ok
```

Here we have two set literals which are combined by the set union operation to form another set as a result. However, because we do not in general know the amount of memory that will be required to represent an arbitrary set, we create sets on the heap, and leave pointers to them on the stack. Thus the `UNION` operation expects two sets on the stack (represented by pointers) and leaves a pointer to the result of the union operation.

Similarly we can calculate the intersection of two sets:

```
{ 1 , 2 } { 2 , 3 } INTER .SET  { 2 } ok
```

The problem of garbage arises because the heap space occupied by sets may still remain allocated after we have finished using them. Simple solutions, such as specifying that set operations release the space occupied by the arguments they consume, are too restrictive because these sets may still be referenced elsewhere. For example consider:

```
{ 1 , 2 } { 2 , 3 } 2DUP UNION -ROT INTER
```

The usual implementation choices for handling garbage are between maintaining reference counts for each set literal, or just maintaining a record of allocations and using a “conservative garbage collector”. Reference counts provide the best memory discipline but complicate the use of simple stack manipulation words like `DUP` on sets, since if a pointer is duplicated we must also increment its reference count. Conservative garbage collection works by scanning memory for references to each allocated region and releasing the region if no references are found. False references may be found, e.g. a literal value which just happens to match an allocated address. The “conservative” aspect of the technique arises from the cautious approach adopted when this occurs.

3 Reversible computation, garbage collection and backtracking

Our virtual machine implementation is capable of reversible execution, by which it can undo the internal effects of previously executed code. To make use of this we augment Forth with two new constructs, choice and guard.

Choice is represented as:
`<CHOICE <operations1> [] <operations2> ...  CHOICE>`
This provides a choice between executing `<operations1>` or `<operations2>` (and possibly other choices).

The guard, represented as `-->` , removes a flag from the stack. If true, execution continues ahead. If false, execution reverses to the most recent choice. If at that point there is still an unused choice, this choice is now made and execution once again continues ahead. If no untried choices remain, backtracking continues to the previous choice, or perhaps to the start of the operation. In the latter case the whole operation is infeasible, and if invoked from the keyboard will provoke a response of "ko" instead of "ok". Other forms of choice are also implemented, e.g. choice of an element from a set.

Reversible execution reduces the problem of garbage collection, and with some programming styles can eliminate it altogether, since during reverse execution any space allocated for heap objects is de-allocated automatically.[2].

4 Implementing Sets

We have experimented with a number of forms of implementation, but here we describe the one in which we have taken the most extreme design decisions. These are minimum concern for garbage generation, limitation to homogeneous sets, and maximum regularity of set representation.

Sets are held as ordered arrays of elements. Each element occupies one cell of memory. In the case of sets of numbers, each elements will just be the actual number represented. In all other cases it will be a pointer to a representation of the element. In addition to a list of elements, each set representation contains a cell which holds a count of the elements, and a cell which holds a comparison function, required to maintain the elements as an ordered array.

5 Technical Notes

Efficiency has been one of the criteria that governed our choice on how sets were represented and implemented on a concrete target machine. In principle, different possible solution for their data representation could have been considered.

5.1 Data Representation of Sets

A naive approach towards representation of sets of arbitrary objects is to utilize linked lists. Whereas enumeration could be done reasonably fast, indexed access would involve iteration through (in the worst case all) list elements. To improve this matter, a second approach may use arrays to represent set objects. This allows indexed access in constant time and moreover enforces a more compact data representation. The crucial disadvantage of both models is the complexity of testing whether an object is contained in a given set or not. Since no assumptions can be made about the order of set elements the complexity of such an operation is of linear order $O(n)$ with respect to the set cardinality. Depending on their precondition, the same overhead arises when invoking the *insert* and *delete* operations, e.g. if we allow to call these operations for objects that may already be contained in the set.

The model that finally has been chosen and implemented represents sets as *ordered arrays*. An ordering of the array elements proves useful as binary search can be applied to determine if an element is contained in the set, and if so, to obtain its position within the array. An implication of this approach is that a comparison function has to be provided to order the elements of a set. The function has to be unique for each possible form of set (i.e. numbers, strings, pairs or sets themselves). Clearly, the function has to impose a total ordering on all potential elements that might be entered into the set.

The following diagram summarises the composition of sets as memory objects:

Comparison Function	4 *Bytes*	*Reference*
Cardinality	4 *Bytes*	*Value*
1. Element (**least**)	4 *Bytes*	*Reference or Value*
2. Element	4 *Bytes*	*Reference or Value*
⋮		
n. Element (**greatest**)	4 *Bytes*	*Reference or Value*

A crucial design decision is reflected by this model, in particular to restrict ourselves to *typed sets*. In *Typed Set Theory* we can assume that all elements of a given set are of the same mathematical type. Therefore, type information only needs be recorded for the set elements as a whole (in form of a comparison function) and not for each element individually.

5.2 Complexity of Set Operations

To determine the index of a given element within the ordered array a logarithmic time complexity $O(log\ n)$ can be assumed. A variant of this operation (with the same complexity) can be created to enquire whether a given element is contained in a set.

The *insert* and *remove* operations first have to determine the position within the array where the element in question has to be inserted. This is necessary since any operation that modifies the set data structure has to maintain its integrity, e.g. the ordering of elements. The second part of both operations moves respective areas of memory, to make space for the new element in case of the *insert* operation or to overwrite the old set element in case of the *remove* operation. Indeed, this is a theoretical weak point of the model since moving memory blocks still demands a complexity of linear order, which yields to a total complexity of $O(n)+O(log\ n) = O(n+log\ n) = O(n)$. However, memory moves are highly optimised operations in advanced processor systems and therefore probably wouldn't make up the bulk of the execution time.

Two important operations on sets are *union* and *intersection*. In our model, these operation become particularly efficient yielding to a complexity of the order $O(n+m)$ if n and m are the cardinalities of both sets. The following algorithm in pseudo code has been formulated to evaluate the intersection of two sets. A similar algorithm can be obtained for set union.

```
int i := 1, j := 1, k := 1;
element1 = {Element at the i-th position of 1st Set};
element2 = {Element at the j-th position of 2nd Set};

while (i <= {Cardinality of 1st Set} and j <= {Cardinality of 2nd Set})
{
    if (element1 = element2)
    {
        {Insert element1 at position k into the resulting Set}
        i := i + 1; j := j + 1; k := k + 1;
        element1 = {Element at the i-th position of 1st Set};
        element2 = {Element at the j-th position of 2nd Set};
    }
    else {
        if (element1 < element2)
        {
            i := i + 1;
            element1 = {Element at the i-th position of 1st Set};
        }
        else {
            j := j + 1;
            element2 = {Element at the j-th position of 2nd Set};
        }
    }
}
```

A question that has been delayed so far is the complexity for comparing two set elements. The previous considerations have been made under the quiet assumption that comparison can be performed in constant time. This can certainly be justified if the set elements are integers. In case of pairs of elements, sets or other types a comparing algorithm has to be designed. We propose the following algorithm to compare two given sets:

- If two sets consist of different numbers of elements, compare them due to their cardinality and return the result.
- If two sets have the same cardinality, compare the last element of the first set with the last element of the second set according to their ordering. If they are **not equal** return the result of the comparison. Otherwise,
- proceed with the second last element of both sets and repeat the last step. The whole process continues until the first element of both sets is reached.

- If consequently the first elements of both sets are equal, return equality of both sets as the result of the comparison.

This algorithm involves recursive invocation if the contained elements of the two given sets to be compared are sets themselves. The complexity is therefore difficult to estimate since it depends on the anatomy of sets used in a particular invocation, e.g. what is the expectation of comparing two sets that have the same cardinality and agree in the last n elements. Our argument for constant complexity of set comparison is based on the assumption that this case is rather unlikely. Indeed, a mathematical elaboration of this problem could be done but shouldn't be the concern of this report.

5.3 Integration of Pairs

Pairs are an essential set theoretical concept which can be used to model associations between data in the form of functions and relations. The integration of pairs into our model will be done in such a way that each element of a pair can itself be of arbitrary complexity. Pairs have to be accessed *by reference* from a given set since their size exceeds the minimum size of 4 byte reserved for one set element. Another problem facing pairs as set elements is that the set itself wouldn't know about the type if each component of a pair object, since this is not apparent from the pair components themselves: further type information has to be incorporated into pair data object. We do this by additional comparison functions. The following diagram illustrates the composition of the pair data structure:

1st Comparison Function	4 *Bytes*	*Reference*
1st Pair Element	4 *Bytes*	*Reference or Value*
2nd Comparison Function	4 *Bytes*	*Reference*
2nd Pair Element	4 *Bytes*	*Reference or Value*

The comparison function for pairs within sets orders pairs according to their first component. If the first components of two pairs are equal, they are ordered by their second component. This has a particular advantage when pairs are used in partial function definitions. Since all maplets are ordered respectively to their domain values in the set representing the function, retrieving the function value can be done as well in logarithmic time by applying binary search to find the index of the according maplet.

5.4 Reversibility of Set Operations

Sets as described so far will be used as part of a reversible machine architecture. This implies that all set modifying operations ought to be reversible. Fortunately, there are only two operation which modify set objects: The *insert* and *delete* operation to insert and remove objects from a given set, respectively. Previously, the complexity of these operations was identified to be determined by a procedure to find the position at which the new element was to be inserted (or removed). In a reversible environment this index is stored on a history stack while the machine runs in forward execution mode. As soon as reverse execution is entered, these operations merely have to fetch the index from the history stack. In case of the delete operation further information has to be kept concerning the element, which was deleted form the set. We can conclude that all set operations have complexities of constant order when reverse execution is enabled.

6 Set Integration into Forth

The model of presentation so far has been quite independent from any particular programming language. Nevertheless, some ideas about how sets can be integrated into a standard Forth system will follow.

The first issue to be considered is memory allocation. Three options are available, namely creating of set objects on the stack, on the heap or within the dictionary memory area of a Forth system. Due to the dynamic nature of set objects, creation on the stack, as well as in the dictionary memory area, has been be excluded. This leaves us with the decision to treat sets as objects in heap memory. The second problem which needs to be solved is the parsing and creation of sets from the source input stream of a Forth system. We consider both of the following alternatives to be feasible:

- Definition within a specialised context of appropriate words for structuring symbols in set expressions, such as " { ", " } " or " , ". The execution time behaviour of those words will perform memory allocations and initialisation of the allocated areas.
- Utilization of a *Set Parser* which is activated upon reading of a " { " bracket and invoked having read the first balanced " } " bracket.

7 Conclusions

Fast processors and large memory space suggest that less efficient but more abstract data representations could be of value in an executable program. Sets are the ultimate choice of abstract data representation. Reversible computing reduces our concern for garbage collection, allowing a very regular representation to be used in which each element occupies one cell of memory. Elements may be numbers, or pointers to more complex forms of element. Our implementation can handles sets of numbers, sets of strings, sets of pairs and sets of sets, where the afore-mentioned pairs and sets may themselves be of arbitrary complexity. The more complex element forms are created on the heap during forward execution and removed during reverse execution. This automatic memory management will be supplemented by a conservative garbage collector.

References

[1] Jean-Raymond Abrial. *The B Book.* Cambridge University Press, 1996.

[2] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in Lecture Notes in Computer Science, 1992. See www.pipeline.com/ hbaker1.

[3] N Bourbaki. *Théorie des ensembles.* Masson, Paris, 1970.

[4] W J Stoddart. A Virtual Machine Architecture for Constriant Based Programming. In P J Knaggs, editor, *16th EuroForth Conference, ISBN 9525310 x x*, Lecture Note in Computer Science, 2000.

[5] W J Stoddart. An Execution Architecture for B-GSL. In Bowen J and Dunne S E, editors, *ZB2000*, Lecture Notes in Computer Science, 2000.