A NOTE ON EFFICIENT CONTEXT SWITCHING

A. Ramsay

Cognitive Science Research Report

Serial no: CSRP 023

The University of Sussex
Cognitive Studies Programme
School of Social Sciences
Falmer
Brighton BN1 9QN

A Note On Efficient Context Switching

Allan Ramsay
Cognitive Studies Programme, University of Sussex, Falmer BN1 9QN
0273 - 606755 X:1008

3500 words, 12 pages

Abstract

It is often convenient to be able to save and subsequently restore the state of a computation. It is also, in general, rather expensive. We present a technique whereby states may be saved and restored extremely fast, and yet the information stored in a given state may be accessed and updated reasonably quickly. This technique depends on an indexing scheme which can be used to see whether the last value assigned to a variable is still valid, or whether it must be retrieved from some previous context (and if so, which one).

1 The Problem

AI programmers, expert system builders, etc. frequently need to write programs that explore large search spaces. Such programs have to save the state of the computation whenever they decide to investigate one area of the search space rather than another, since they may make the wrong choice and be forced to back up and try again.

The process of saving and restoring computational states is expensive in two ways. It requires a large amount of memory (at worst a complete copy of the process image for every saved state), and it takes time. The costs may be diminished if we are content to save less than the entire state, e.g. just saving the calling sequence, or just saving the values of some predetermined set of variables. We are mainly concerned here with this latter case, where the values of a specified set of global variables are to be saved and restored (this restricted form of state saving has been widely used in AI - see for instance Kaplan's discussion of ATN parsing for natural language (1), or CONNIVER's contexts (2)). However, the mechanism described below will deal with arbitrary cases of variable access and assignment, and could easily be adapted for contexts which included the values of local variables as well as globals.

A number of implementations of this restricted form of state saving have been proposed, but they have generally suffered either from taking a long time to switch contexts, or from taking a long time to access the values of variables within a given context. We present below a technique for context switching in which contexts may be saved and restored extremely quickly, and yet on most occasions the value of a variable may also be accessed without any search.

2 Contexts vs. Stack Frames

The discussion of how to implement a state saving mechanism becomes clearer if we relate our problem to that of saving and restoring the values of dynamically bound local variables during procedure calls. The effect of such a procedure call is to save the current values of the local variables, i.e. the current context; do whatever processing is required; and then restore the saved context. This is exactly the problem we are dealing with, except that the contexts saved by procedure

calling mechanisms are always dealt with first-in first-out, whereas we want to be able to restore arbitrary contexts, e.g. ones that were saved after the one that is current, or ones that have been restored once already but that we want to try yet again.

Procedure calling mechanisms save the current context by stacking the values that are to be saved, and popping them when they are to be restored. We cannot use this approach, since we cannot rely on contexts being saved and restored in any particular order. We can, however, begin to solve our problem by looking carefully at two strategies that are widely used for binding values to variables, namely deep and shallow binding.

Allen provides a detailed discussion of these strategies (3); all we will do here is to summarise the main points as they relate to our particular problem.

In deep binding, all values of variables are kept on a single stack of variable-value pairs. Assignment is done by pushing a new pair, containing the relevant variable and its new value, onto the stack; variable access is done by walking down the stack until a pair containing the required variable is found - the value in this pair is the most recent, i.e. current, value of the variable; saving and restoring contexts is done by saving the position of the top of the stack to save the context, resetting it to that position to restore it. Thus in this very pure form of deep binding, context switching is extremely easy, assignment is easy but wasteful of space, and access is unacceptably slow.

In shallow binding, each variable has a stack of values directly associated with it. The current value of a variable may be accessed very quickly, since it is simply the top element of the associated stack, and it may be updated by overwriting that element (not by pushing it on top). However, context switching is rather laborious, since it is necessary to scan all the variables whose values have to be saved or restored and either push a copy of their current value onto their associated stack (saving) or pop the current value off (restoring).

In practice most implementations use a combination of these strategies, keeping saved values on a single stack and current values associated directly with the variables (see for instance UCILISP (4) or POPL06 (5)). In the remainder of this paper we will adapt this technique for context switching where the simple FIFO constraint of procedure calling is not obeyed.

5 Full Context Switching

For our task it is convenient to adapt the basic strategies as follows:

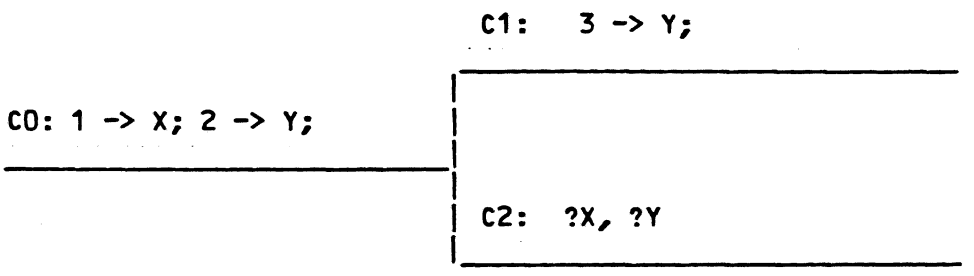
(i) a context is still a stack of variable-value pairs (a binding stack), but it now has associated with it an index. We can use the indices to see whether one context is a direct descendant of another.

(ii) when we update a global variable we add a new variable-value pair to the current stack, as in deep binding; and we associate the new value directly with the variable, as in shallow binding; but we also associate the index of the current context with the variable, so that we know when the value was set.

(iii) to save a context, we now simply save the binding stack and its index; to restore it, we reset the saved stack and index to be the current stack and index. Thus all we need to do when we want to switch contexts is to update the values of two global variables - there is no need to construct any new data structures or to transfer large blocks of data.

(iv) variable access is a little more complex. The essential point is to compare the index of the current context with the index that was stored with the variable when it was last updated, using an algorithm described below. to see if the variable's index is a direct predecessor of the current one. If it is then the value associated directly with the variable must be the required one, since the context in which it was last updated is a direct ancestor of the current one. If, on the other hand, the index for the variable is not a predecessor of the current one, then the variable must have been updated on some divergent branch of the computation. In this case we will have to search down the current binding stack to find the value we want.

To see what is going on, consider the following example:



The above diagram represents a situation where 1 has been assigned to X and 2 to Y in context C0; C0 has been saved, and continued as C1, and 3 has been assigned to Y; and finally C1 has been saved, and C0 has been restarted as C2. What are the current values of X and Y ?

Clearly the value directly associated with X, i.e. the value most recently assigned to it, is 1. This value was assigned in C0, which is a direct ancestor of the current context, and hence is the value we want.

The value currently associated with Y is 3, the value assigned in C1. C1 is not a direct ancestor of C2, so this value is not correct in C2. Hence we will have to search down the the binding stack for C2 for the required entry. C2's stack is built on top of C0's, so the first value we will come to is 2, as required.

This all sounds very long-winded. The crucial point is the construction and comparison of indices, since if this can be done quickly then in many cases we can access variables almost as fast as if they were shallow bound in an environment with simple FIFO context switching, and we will never do worse than orthodox deep binding.

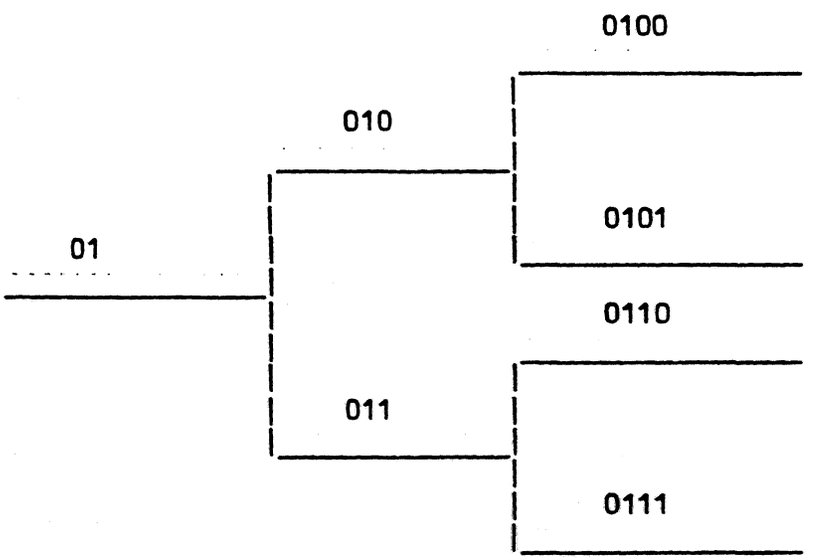
4 Indexing Contexts

Suppose we are just starting work, so the stack is empty, the index 1, and all the variables are initialised with the standard v "undefined" and index 1. We do some work, updating the stack and variables as we go, and eventually arrive at a decision point, where want to save the current state of affairs for later while we explo

some hypothesis.

At this point we need to create two contexts, namely one to save for later and one to make immediate use of. For each we initialise the stack as a list containing nothing but a pointer to the current binding stack, so that the binding stack in any given context is in fact a chain of partial stacks. We obtain their indices by doubling the current index, and adding 1 to it for one of them and not for the other. At this point we have created two contexts, with stacks equivalent to the original and indices 010 and 011 (binary), simply by copying a pointer and doing some trivial arithmetic.

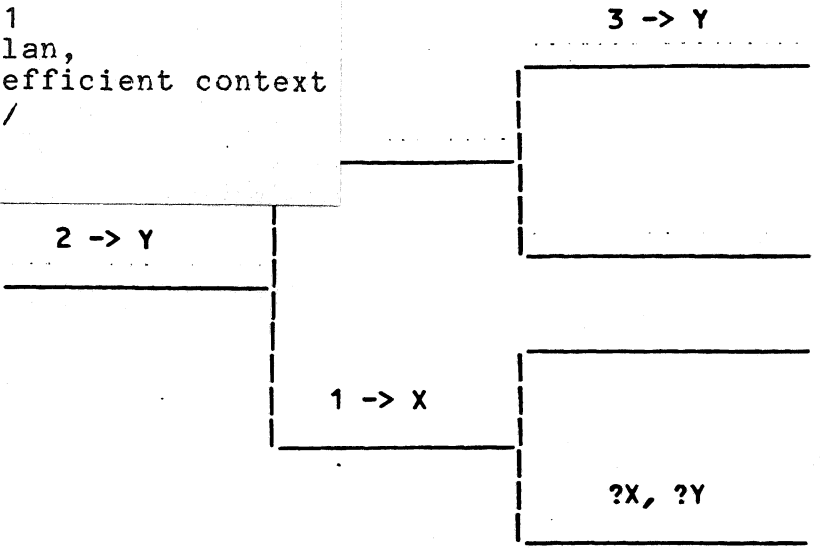
As we continue to spawn contexts we will grow a tree of them with indices marked as below



It is now easy to see if one index, I1 say, is a direct predecessor of another, I2. We simply keep comparing them and then shifting I2 right one place until either they are equal (I1 is a direct predecessor of I2) or I2 is less than I1 (I1 is not a direct predecessor of I2).

This gives us what we needed - a rapid algorithm for generating indices (shift the old one left one place, use this for the state which is to be saved and add 1 for the one which is to be continued), and a rapid algorithm for comparing indices (repeatedly shift right and compare).

To see again how this works out, suppose that as we grew the above tree we had done the following assignments to X and Y.



In context 0100, the chain of partial binding stacks would have looked like

```
Stack for 0100      Stack for 010      Stack for 01
    [Y 3] ----------> [] ----------> [Y 2]
```

whereas in context 0111, it would have looked like

```
Stack for 0111      Stack for 011      Stack for 01
    [] ---------------> [X 2] --------> [Y 2]
```

What are the values of X and Y in context 0111 ?

X has associated value 1, associated index 011, and the current index is 0111. We compare the the two indices; the current one is greater, so we shift it right and it becomes 011; this is equal to the one associated with X, so we know that the value 1 associated directly with X is indeed its current value.

Y has associated value 3, associated index 0100. We compare this index with the current one; they are not equal, and the current one is greater than Y's, so we shift it right and compare them again; the shifted index is now 011, which is less than Y's index, so we know that the value associated with Y is no longer valid and we will have to search the current binding stack. This includes the stack containing the entry for 2 -> Y, so we will find the required value .

5 Refinements

There are a number of refinements to the above scheme which will improve its performance without altering its essential properties, as follows;

(i) if you have to search the stack for the value of a variable, it is probably worth updating its associated value and index to the value you have just found and the current index. If you have just accessed a variable in a given context, you are more likely to access it next in the same one or some direct descendant of it than in any one other context, so it makes sense to ensure that you will be able to find the value directly on this path rather than on the one you have just

switched from.

<ii) in the discussion above, we said that every time you update a variable you have to push a new entry onto the binding stack. In fact, if you update a variable twice in the same context, the second entry will permanently obscure the first. The first entry willthen simply waste space and make searches of the stack for other variables take longer. For this reason, instead of associating the current value itself with the variable we associate the current stack entry. This extra level of indirection has little effect on the access time, and allows us to update the stack entry rather than adding a new one whenever we repeatedly assign to a variable in the same context - a particularly important consideration in view of the effects of a fragment of code such as

```
repeat 100 times 1+1 -> I; ... endrepeat;
```

which would otherwise add 99 unwanted entries to the binding stack.

These refinements to the basic scheme can each be implemented without introducing any change in its functional behaviour. If they are both to be used within the same system, it is essential to note that entries created by virtue of refinement (i) should not be overwritten when the variable is next assigned, as described in refinement (ii).

£ Conclusions

The mechanism outlined above for saving and restoring arbitrary contexts compares very favourably with previous systems with the same goal (6, 7). Saving and restoring contexts becomes a trivial matter; assigning values to variables takes a short, nearly fixed amount of time, as does accessing their values except when you have switched into a context which is incompatible with the one in which the variable was last accessed or assigned. Admittedly, we require more space for storing saved values than would be necessary if contexts were always switched under a FIFO discipline. We do, however, make reasonably economic use of space by ensuring that every context is built directly on top of its actual predecessor, rather than on a copy of it, so that we do not store unnecessary copies of information.

The mechanism has been implemented in P0P-11, but only to save and restore the values of a user-specified set of global variables. As such it has been used effectively to implement the "registers" of a back-tracking ATN parser (1). It does, however, provide a complete description of variable access and assignment, and hence could easily be used for the temporary assignment of local variables during procedure calls. Using the mechanism described here would inevitably be slower than the techniques currently used, but it would greatly facilitate the implementation of the state-saving required for more complex control structures than simple call-and-return.

References

(1) R.M. Kaplan A general Syntactic Processor in "Natural Language Processing", ed.l"- Rustin, Algorithmic Press 11973)

(2) G.J.Sussman & D.V.HcDer»ott CONNIVER Reference Hanual MIT AI Pe»o

259 (1972)

(3) J. Allen Anatomy of LISP McGraw-Hill (1978)

(4) J.R. Meehan New UCI LISP Manual Lawrence Erlbaum Associates (1979)

(5) S. Hardy The POPLOG Programming System Cognitive Studies Research Paper 3, Univ. of Sussex (1982)

(6) D.G.Bobrow & B.Wegbreit J* Model For Control Structures For AI Programming Languages IEEE Trans, on Computers (1976)

(7) J. Urmi A Shallow Binding Scheme For fast Environment Changing In J\ "Spaghetti TtacF[1] TTSP System" CTTTRTAT-R-76-18, Drvrv. of Linkoeping (1976)