ELSEVIER

Information Processing Letters 52 (1994) 15–22

Information Processing Letters

WORM-2DPDAs: An extension to 2DPDAs that can be simulated in linear time

Torben Æ. Mogensen [1]

DIKU, Universitesparken 1, DK-2100 Copenhagen O, Denmark

Communicated by R. Bird; received 11 April 1994; revised 20 June 1994

Abstract

We extend 2-way deterministic push-down automata (2DPDAs) with a write-once-read-many (WORM) store. We show that it allows linear time simulation by a variant of Cook's construction. As an example we develop a linear time algorithm that recognizes the language $\{W^{-1}WW^{-1} \mid V, W \in (a \mid b)^*\}$, that by Aho, Hopcroft and Ullman is conjectured not to be recognizable by a 2DPDA. Thus we believe that the extension strictly increases the expressive power of 2DPDAs.

Keywords: Algorithms; Automata; Linear time simulation; Program derivation

1. Introduction

Two-way deterministic push-down automata (2DPDAs) have played an important role in algorithm design, due to a result by Cook [3] that shows that any algorithm expressed as a 2DPDA can be executed in linear time on a RAM by means of memorization tables. The advantage of using 2DPDAs in algorithm design is that it is often easier to find a 2DPDA solving a problem than to write a linear time algorithm directly.

Jones [4] modifies Cook's original construction to calculate the memorization tables on-line. The on-line construction allows us to extend 2DPDAs with a limited kind of storage, in addition to the stack, without losing the linear time simulation.

In [2], Andersen and Jones extend Jones' work by expressing the simulation as a compilation process, thereby eliminating most of the interpretative overhead of the earlier construction. We will use this transformation as the basis for our own, as well as adopting their more free-form program notation for 2DPDAs.

2. WORM-2DPDAs

A traditional 2DPDA consists of an input tape, a stack and a set of states. Each state can, depending on the symbol on the stack top and the symbol at the tape position, pop or push a symbol on the stack, move the tape one step left or right and go to another state. Two selected states stop the machine with, respectively, success or failure.

Various extensions to the basic 2DPDA have been studied. Allowing more freedom in moving the tape position adds no extra power, and is often implicitly allowed when expressing algorithms as 2DPDAs. Adding extra tapes and/or extra tape-position pointers does increase the power, but also increase the time

[1] Email: torbenm@diku.dk.

SSDI 0020-0190(94)00118-I

of simulation. Two tapes/pointers increases the simulation time to quadratic, etc.

We will add a limited kind of modifiable store (in addition to the stack) and argue that this does not affect the linear time simulation property. In fact, the extra store can essentially be ignored by the simulation/transformation. The idea is as follows:

In addition to the components of a 2DPDA, a WORM-2DPDA consists of any finite number of WORM-tapes parallel to the input tape. The WORM tapes are not accessed by separate position pointers, but share the pointer that is used to access the input tape. As the name indicates, each position on the WORM tape may be written to once only, but may be read from several times. The exact restrictions that apply are crucial to preserving the linear time simulation:

(1) Initially all positions on the WORM tapes are undefined.
(2) Writing a symbol to an undefined position defines it to contain that symbol.
(3) Reading from an undefined position immediately makes the position defined to contain an (arbitrary) default symbol.
(4) Writing to a defined position has no effect, i.e., the symbol at the position does not change.
(5) Reading from a defined position (of course) returns the symbol at the position.

The important invariant is that *if a position is read twice, the same symbol is returned, no matter what has happened before or between the two* read *operations.* Also, it is impossible (from within the machine itself) to determine if a WORM tape position is undefined.

Restriction (3) can be replaced by a requirement that all reads to a WORM tape must be immediately preceded by a write to the same position. Alternatively, reading from an undefined position could cause an abnormal termination of the automaton (a "run-time error"). What matters is that the invariant above holds.

2.1. Program notation

We describe a WORM-2DPDA by a set of labeled statement blocks. The syntax of a statement block is shown in Fig. 1. Each block consists of a statement structure, with jumps or `end` statements at the leaves. The variable `i` is the tape position pointer, which is used to index both the input tape and the WORM tapes.

```
Block     → label : Statement
          | label : pop; Statement

Statement → i := Iexp; Statement
          | push Sexp; Statement
          | if Bexp then Statement else Statement
          | worm_j [Iexp] := Wexp; Statement
          | end Bexp
          | goto label
```

Fig. 1. Syntax of a statement block.

We allow arbitrary index calculations depending on `i`, the tape length `n`, the stack top `top`, and all tape symbols. *Iexp* is an expression that returns an index. The index `i` is constrained to be between 0 and `n` − 1. *Sexp* is an expression that returns a symbol from the stack alphabet Σ. *Wexp* is an expression that returns a value in the alphabet used for the WORM tapes. *Bexp* is an expression that returns a truth value, e.g. `tape[i] = 'a'`. We place no constraints on the forms of these expressions, but note that some expressions may be impossible to simulate on traditional automata. In particular, we allow infinite alphabets for the input tape and the WORM tapes. This allows, for example, indexes to be stored on the WORM tapes. The linear time simulation result is independent of which restrictions we place on the expressions, as long as all expressions evaluate in constant time. We need no restriction on the range of expressions, except where the result is stored in `i` or pushed on the stack. We can even allow the WORM tapes to be of arbitrary length, as the range of indexes used to address these do not affect the complexity of the simulation.

A restriction which will be important for the transformation, is that `pop` statements are labelled, i.e. they are the first statements in their statement blocks. Also (a point not shown in the grammar), blocks with pops cannot contain `push` statements, and no block will contain more than one `push` statement.

There is no room here for a detailed formal semantics of the language shown. But, apart from operations on the WORM tape, program execution is straightforward. Regarding the WORM tapes, one just needs to take into account the properties stated in Section 2.

An example of a WORM-2DPDA is shown in Fig. 2. It determines if a string consists of exactly two palindromes, or more precisely if the string is

```
start:      push tape[i];  i := i+1;
            if i<n-1 then goto start
            else i := 1;  goto loop1

loop1:      if top='>' then
                 worm1[i-1] := '1';
                 if i=1 then i := n-1;  goto start2
                 else goto restore1
            else if top = tape[i] then goto continue1
                 else goto restore1

continue1:  pop;  i := i+1;  goto loop1

restore1:   i := i-1;
            if i=0 then goto continue1
            else push tape[i];  goto restore1

start2:     push tape[i];  i := i-1;
            if i>0 then goto start2
            else i := n-2;  goto loop2

loop2:      if top='<' then
                 if even(i) and even(n) and worm1[i]='1' then
                      end true
                 else if i=n-2 then end false
                      else goto restore2
            else if top = tape[i] then goto continue2
                 else goto restore2

continue2:  pop;  i := i-1;  goto loop2

restore2:   i := i+1;
            if i=n-1 then goto continue2
            else push tape[i];  goto restore2
```

Fig. 2. A WORM-2DPDA recognizing $\{W^{-1}WW^{-1} \mid V, W \in \Sigma^*\}$.

in the language $\{W^{-1}WW^{-1} \mid V, W \in \Sigma^*\}$. There is one WORM tape called `worm1`. First, all palindromic prefixes are recorded on the tape (by writing 1's at the relevant positions); then palindromic suffixes are found. If a palindromic suffix is adjacent to a palindromic prefix at an even tape position (and `n` is even), the string consists of two even length palindromes. The part of the WORM-2DPDA that finds all palindromic prefixes is adapted from a 2DPDA in [2], that find the longest palindromic prefix. The part that finds palindromic suffixes is a simple modification of this. The end markers on the input tape are > and <, at positions 0 and $n-1$. `i` is initialized to 0. We assume 0 is written on undefined positions on the WORM tape, if they are read from before being defined. The input alphabet is arbitrary, as only equality tests are used. By removing the tests on the evenness of `i` and `n`, the automaton will recognize strings consisting of two arbitrary palindromes.

2.2. *WORM-2DPDAs are stronger than 2DPDAs*

The WORM-2DPDA in Fig. 2 is the basis of our argument that WORM-2DPDAs are stronger than traditional 2DPDAs. If we believe the conjecture in [1] about the inability of any 2DPDA to recognize the language $\{W^{-1}WW^{-1} \mid V, W \in (a \mid b)^*\}$, we must believe that WORM-2DPDAs are stronger than 2DPDAs. Note that, even though we have used arbitrary index calculations, etc., those used in the example are

$\mathcal{B}$: *Block* → *Block'*

$\mathcal{B}[\![l$: `if dest[`l`,i,top] is defined then`
`i := snd(dest[`l`,i,top]); goto fst(dest[`l`,i,top])`
`else dumptop := (`l`,i) :: dumptop;` $\mathcal{S}[\![s]\!]$

$\mathcal{B}[\![l$: `pop;` $s]\!]$ = l : `for all (`l`',i') in dumptop do dest[`l`',i',top] := (`l`,i);`
`pop; popdump;` $\mathcal{S}[\![s]\!]$

$\mathcal{S}$: *Statement* → *Statement'*

$\mathcal{S}[\![$`i := `e`;` $s]\!]$ = `i := `e`;` $\mathcal{S}[\![s]\!]$

$\mathcal{S}[\![$`push `e`;` $s]\!]$ = `push `e`; pushdump emptylist;` $\mathcal{S}[\![s]\!]$

$\mathcal{S}[\![$`if `e` then `s_1` else `$s_2]\!]$ = `if `e` then` $\mathcal{S}[\![s_1]\!]$ `else` $\mathcal{S}[\![s_2]\!]$

$\mathcal{S}[\![$`worm`$_j$`[`e_1`] := `e_2`;` $s]\!]$ = `worm`$_j$`[`e_1`] := `e_2`;` $\mathcal{S}[\![s]\!]$

$\mathcal{S}[\![$`end `$e]\!]$ = `end `e

$\mathcal{S}[\![$`goto `$l]\!]$ = `goto `l

Fig. 3. Translation of WORM-2DPDAs to flow-chart programs.

easily representable using the traditional restrictions on two-way automata.

3. Linear time simulation

Following the lead of Andersen and Jones [2], we present the simulation by a transformation that will convert any WORM-2DPDA to a linear time flow-chart program.

We add two extra components to the machine: a *dump* and a *destination table*. The dump is a stack of lists of pairs, each pair consisting of a label and an index value. The dump grows and shrinks in parallel with the normal stack, but the list on top of the dump can grow until it is popped off. The destination table is indexed by triples of labels, indexes and stack-alphabet symbols. The entries are initially undefined, but may be updated to contain label/index pairs. In addition to suitable operations on the dump and destination table, we extend the machine with indirect jumps.

The transformation is shown in Fig. 3. Fig. 4 shows the transformed version of the program from Fig. 2. We have made some simplifications which are discussed below.

The principle is the same as in the classic construction: if we enter a program point for a second time with the same values of the index and the stack-top symbol, we will repeat the same steps until the point at which we pop the stack. By remembering these steps, we can, on later occurrences of the same situation, skip directly to the state just before the pop. The destination table will, the first time round, be updated to contain a description of this situation, and this will be used the second (and all subsequent) times to go directly to the pop. The dump is used to record all the surface states (label, index and stack-top symbol) we have been in since the last push, so we can update the relevant parts of the destination table when we pop the stack.

Note that, while the program flow depends on the contents of the WORM tapes, this dependence is exactly repeated the second time we are in the same surface state. This is because we will read the same positions on the WORM tapes and get the same results back (the invariant mentioned in Section 2). As the machine is deterministic, we cannot deviate from the path we took the first time. But this path can certainly depend on what was written to the tape earlier. This dependence necessitates the use of an *on-line* construction of the destination table, as in Andersen and Jones' work [2,4], as opposed to the *off-line* construction in Cook's original paper [3].

We will informally argue that the transformation is correct and that the transformed program runs in linear time. A somewhat more formal proof of Jones' variant of Cook's construction can be found in [2]. The proof can be used essentially unchanged for WORM-2DPDAs. A paper in preparation by the author will address the correctness of the transformation relative to formal semantics of the two languages involved.

First we argue that the transformation does not change the final result of the automata. This is essentially what we did above, assuming we believe the handling of the dump and destination table is correct.

```
start:      push tape[i]; pushdump emptylist; i := i+1;
            if i<n-1 then goto start
            else i := 1; goto loop1

loop1:      if top='>' then
                worm1[i-1] := '1';
                if i=1 then i := n-1; goto start2
                else goto restore1
            else if top = tape[i] then goto continue1
                else goto restore1

continue1:  for all (l',i') in dumptop do dest[l',i',top] := (continue1,i)
            pop; popdump; i := i+1; goto loop1

restore1:   if dest[restore1,i,top] is defined then
                i := snd(dest[restore1,i,top]); goto fst(dest[restore1,i,top])
            else
                dumptop := (restore1,i) :: dumptop;
                i := i-1;
                if i=0 then goto continue1
                else push tape[i]; pushdump emptylist; goto restore1

start2:     push tape[i]; pushdump emptylist; i := i-1;
            if i>0 then goto start2
            else i := n-2; goto loop2

loop2:      if top='<' then
                if even(i) and even(n) and worm1[i]='1' then
                    end true
                else if i=n-2 then end false
                    else goto restore2
            else if top = tape[i] then goto continue2
                else goto restore2

continue2:  for all (l',i') in dumptop do dest[l',i',top] := (continue2,i)
            pop; popdump; i := i-1; goto loop2

restore2:   if dest[restore2,i,top] is defined then
                i := snd(dest[restore2,i,top]); goto fst(dest[restore2,i,top])
            else
                dumptop := (restore2,i) :: dumptop;
                i := i+1;
                if i=n-1 then goto continue2
                else push tape[i]; pushdump emptylist; goto restore2
```

Fig. 4. A linear time program recognizing $\{VW^{-1}WW^{-1} \mid V, W \in \Sigma^*\}$.

Initially the destination table is everywhere undefined, so until we update it we will follow the original program exactly, with the exception of doing a bit of extra work. This extra work updates the dump such that the list at a certain stack element contains all the surface states (minus the stack-top symbol, which can be found in the stack) that we have visited since the corresponding stack symbol was pushed and while this was the stack top. When we get to a pop, the top of the dump thus contains all surface states we have been in which leads to this pop. The destination table is now updated so all these surface states "point" to the surface state just before the pop. This will allow jumps from these surface states to the pop state via the destination table. But as this state would eventually be reached through normal execution anyway, no change to the final result is made. Any writes that would have been made to the WORM tapes have already been done on the first time through (before the destination table was updated), so there is no need to repeat them.

As for linear time execution, we will argue that the number of execution steps is bounded linearly by the number of surface states encountered during execution. Since the stack alphabet and the number of labels are both of constant size, this implies that execution time is linear in the range of the index and hence the size of the input tape.

We distinguish between blocks starting by pop and blocks without pop.

When we reach a given block without pop for a second time, with the same surface state, there are two cases: either the symbol that was on the stack top the first time through is still somewhere in the stack, or it has been popped off (and possibly replaced with other symbols). In the first case, the control flow from the first entry to the second entry does not depend on anything below the top of the stack. Since the new stack top is the same, the exact same control flow will be repeated ad infinitum: we have entered an infinite loop. In the case where the symbol has been popped off, the destination-table entry for that surface state has been defined, and the body of the block will not be entered. Instead, we jump to the block that performed the pop.

Hence, if the program terminates, each block without a pop is executed at most once for each surface state. Any subsequent time, the destination table will be defined. Thus, the total number of pushes to the stack will be bounded by the number of surface states (there is at most one push in a block). By a similar reasoning, each surface state will only be added to the dump once. So the accumulated cost of updating the destination table in blocks with pop is linear in the number of surface states. We also note that the second (or subsequent) time a surface state is entered, it will immediately do a pop (either because the block starts with a pop, or because we used the destination table to jump to such a block). Since the total number of pushes was bounded by the number of surface states, the total number of "second visits" to surface states will also be thus bounded, otherwise we would pop from an empty stack.

We can, with a simple modification of the linearizing transformation, also get linear time execution in the cases where the original program does not terminate. Any infinite loop must enter some surface state repeatedly. If the destination table has been defined between the two visits, the block in question is not entered and a pop is done shortly afterwards. So only if the destination table is not defined is there a risk of nontermination. If the destination table is not defined, it means that the stack contents at the time of the first entry have not been touched (if they had, the destination table would be defined when the top was popped off). So the actions performed between the two visits can not depend on anything below the top of the stack, and will hence be repeated after the second visit, and so on ad infinitum. We must thus be able to detect this situation, and stop when it occurs. We add a new value *visited*, that an entry in the destination table can take. When we enter a block without a pop, we test the entry in the destination table. If it is *undefined* we set it to *visited*, update the dump, and enter the body of the block. If it is *visited*, we stop with an error message; and if it is defined to be a label/index pair, we set the index and jump to the label. The current implementation does not handle nontermination, as we want to keep the overhead down, and because we are mainly interested in strongly terminating programs.

3.1. Simplifications

As in the paper by Andersen and Jones [2], we can make a few simplifications to the transformed programs. If the program contains blocks that we are sure will only be entered once for each index, then there

Table 1
Test runs

length	$p(a)$				
	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{128}$	0
32	0	196	705	937	1000
	0.0s	0.1s	0.1s	0.1s	0.1s
	0.1s	0.1s	0.1s	0.1s	0.1s
64	0	40	448	846	1000
	0.0s	0.2s	0.3s	0.4s	0.4s
	0.1s	0.2s	0.2s	0.2s	0.2s
128	0	1	257	721	1000
	0.1s	0.3s	1.1s	1.5s	1.5s
	0.2s	0.3s	0.4s	0.4s	0.4s
256	0	0	53	532	1000
	0.2s	0.7s	2.9s	5.3s	5.9s
	0.4s	0.4s	0.7s	0.8s	0.7s
512	0	0	2	269	1000
	0.6s	1.4s	6.0s	17.7s	23.6s
	0.7s	0.8s	1.0s	1.5s	1.4s
1024	0	0	0	54	1000
	1.2s	2.8s	11.8s	48.9s	93.1s
	1.4s	1.3s	1.7s	2.5s	2.7s
2048	0	0	0	4	1000
	2.3s	5.8s	24.0s	100.2s	374.4s
	2.6s	2.7s	3.2s	4.0s	5.4s

is no need to use the destination table or update the dump. In the example program, `start` and `start2` are examples of this. Also, if all paths from a block to itself pass through blocks that use the destination table, or contain pop without any intervening `push`, we can omit the extra code for that block. This is true for the blocks labeled `loop1` and `loop2` in the example.

4. Implementation

The transformation has been implemented by a system that compiles WORM-2DPDAs to C, both with and without the linearizing optimization. The simplifications discussed in Section 3.1 have not been completely implemented: the analysis that determines if it is safe to omit the extra code is not implemented, but the user can annotate blocks that he is certain will not need the code, and the transformation will not insert it.

The example in Figs. 2 and 4 has been run on a large number of random strings. This may not be the most appropriate test data, as a truly random string with equal probability of a's and b's will be rejected in average linear time by the unoptimized WORM-2DPDA. As a consequence, several sets of test data were generated with various probabilities of a's and b's. Table 1 shows the result. On the horizontal axis, $p(a)$, the probability of any character being an a is varied. On the vertical axis the length of the string is varied. In each entry three numbers are shown. The first is the number of strings that are in the language recognized by the WORM-2DPDA, the second is the running time for a 1000 random runs with the unoptimized automata and the third is the running time for the same runs with the optimized program. When $p(a) = \frac{1}{2}$, the average number of comparisons needed for the naive algorithm for a string of length n is $2n$, so the linearization does not improve the average asymptotic complexity. The interesting thing is that the overhead of updating and using the destination table is relatively small. When $p(a)$ decreases, the transformed program quickly wins over the naive algorithm. When $p(a) = 0$, the naive algorithms clearly shows $O(n^2)$ behaviour while the optimized program retains $O(n)$ complexity.

5. Variations

The important property of the WORM tapes is that changes to them cannot be detected from within the automata itself. Any store with this property can be added to a 2DPDA without losing the linear time simulation property. A variation of WORM tapes require writes to be at the end of the tapes, hence require no index for write operations. Reads from the tapes do require an index, and if you read beyond the end of the tape, either an error occurs, or (as in the WORM-2DPDA above) a default value is written. This default value will have to be written to all positions on the tape, from the current end up to the point where the read was attempted, which becomes the new end. The linearizing transformation has to be slightly modified to record the write operations that happens between a push and a pop. The transformed program will run in time bounded linearly by the maximum of the sizes of the input tape and the WORM tapes.

Another variation does away with the execution time's dependence on the size of the stack alphabet, at the cost of restriction to the automata: the stack top cannot be inspected without popping it. This makes pop an expression, which can be used for calculating indexes or values to write to the WORM tapes. The point is that the stack top symbol no longer needs to

be a part of the surface state, since the control up to the next pop does not depend on it. The restriction does make it harder to write some programs, though. The independence of the automaton in Fig. 2 on the stack alphabet cannot (in an obvious way) be maintained, making the size of the automaton (and hence the simulation time) dependent on the alphabet.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, Time and space complexity of pushdown automaton languages, *Inform. and Control* **13** (3) (1968) 186–206.

[2] N. Andersen and N.D. Jones, Generalizing Cook's transformation to imperative stack programs, in: *Results and Trends in Theoretical Computer Science*, Lecture Notes in Computer Science **812** (Springer, Berlin, 1994) 1–18.

[3] S.A. Cook, Linear-time simulation of deterministic two-way pushdown automata, in *Information Processing 71* (North-Holland, Amsterdam, 1972) 75–80.

[4] N.D. Jones, A note on linear time simulation of deterministic two-way pushdown automata, *Inform. Process. Lett.* **6** (4) (1977) 110–112.