1-1-1980

Comparative Schematology and Pebbling with Auxiliary Pushdowns

Nicholas J. Pippenger
Harvey Mudd College

COMPARATIVE SCHEMATOLOGY AND PEBBLING WITH AUXILIARY PUSHDOWNS

(Preliminary Version)

Nicholas Pippenger
Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract: This paper has three claims to interest. First, it combines comparative schematology with complexity theory. This combination is capable of distinguishing among Strong's "languages of maximal power," a distinction not possible when comparative schematology is based on computability considerations alone, and it is capable of establishing exponential disparities in running times, a capability not currently possessed by complexity theory alone.

Secondly, this paper inaugurates the study of pebbling with auxiliary pushdowns, which bears to plain pebbling the same relationship as Cook's study of space-bounded machines with auxiliary pushdowns bears to plain space-bounded machines. This extension of pebbling serves as the key to the problems of comparative schematology mentioned above. Finally, this paper advantageously displays the virtues of recent work by Gabber and Galil giving explicit constructions for certain graphs, for the availability of such explicit constructions is essential to the results of this paper.

1. Introduction

To explain why some programming languages are "more powerful" than others is of obvious importance, both for the understanding of current languages and for the design of future ones. It is difficult, however, to say precisely what is or should be meant by "more powerful" in this context.

Early workers recognized that very rudimentary features (such as the ability to maintain several counters) gave any language possessing them the ability to express all the partial recursive functions, thus making all such languages equivalent as regards the functions they can express.

An avenue of escape from this difficulty was opened by Paterson and Hewitt in their paper "Comparative Schematology" [5]. By dealing with the computation of functionals rather than merely with the computation of functions, it is possible to show, for example, that programs with recursive subroutines are more powerful than programs without them, in the sense that there are functionals expressible by the former but not by the latter. This work loosed a flood of papers in which the expressive power of various sets of programming language features were compared.

It was soon recognized, however, that many distinctions that were clear from an intuitive point of view still could not be made precise in this way. Strong, in his paper "High Level Languages of Maximum Power" [9], pointed out the existence of a class of functionals, which he termed the "effective functionals," that plays a role analogous to that of the partial recursive functions. Again it happens that very rudimentary features (such as the ability to maintain several counters and a stack of domain values) give any language possessing them the ability to express all the effective functionals, thus again making all such languages equivalent.

Another possible avenue of escape lies in the consideration of the complexity of functions rather than merely their computability. This approach is implicit in many results concerning efficient simulations between machine models (since a machine model can be regarded without much effort as a programming language). It is easy to show, for example, that there can be no efficient on-line simulation of machines with tapes by machines with counters. This and similar results (which invariably restrict consideration to on-line or even to real-time simulations) have obvious interpretations in terms of programming languages that provide various data types and data structuring facilities.

The present state of affairs in complexity theory, however, does not allow even so gross a distinction as the one between tapes and counters to be made without the restriction to on-line computations. To show, for example, that there is no efficient off-line simulation of machines with tapes by machines with counters would imply (if "efficient" means "preserving polynomial time bounds") that polynomial time is not included in logarithmic space, long an outstanding open problem.

In this paper we shall combine comparative schematology with complexity theory to obtain results that are beyond the reach of either approach alone. To our knowledge, this combination has been used only once before: Paterson and Hewitt [5] showed that although programs without recursion can simulate programs with linear recursion, they cannot do so in linear time. The combination appears promising with regard to many problems concerning programming languages, but attention will be confined here to a single example of its fruitfulness: we shall show that programs with coroutines are more powerful than programs with subroutines. We shall assume that both the subroutines and the coroutines can perform fully interpreted computations (say, arithmetic and logical operations) at reasonable cost, and that subroutines and coroutines may be recursive. Under these assumptions, programs with subroutines can compute all effective functionals, so there is no hope of showing that coroutines are more powerful on the basis of computability considerations. We shall show, however, that although programs with subroutines can simulate programs with coroutines, they cannot do so in polynomial time.

To prove the result just stated, this paper introduces the notion of *pebbling with auxiliary pushdowns*, which bears to ordinary pebbling (see Paterson and Hewitt [5]) the same relationship that space-bounded machines with auxiliary pushdowns (see Cook [1]) bear to ordinary space-bounded machines. In both situations with auxiliary pushdowns there are additional storage media that can only be manipulated according to a restricted discipline but which provide storage not counted in space bounds.

Finally, the graphs that we consider pebbling are constructed using recent results of Gabber and Galil [2]. Our results make essential use of the fact that they provide explicit constructions for graphs that were previously known to exist only through inexplicit probabilistic or counting arguments. The present paper appears to provide the first example of a result that can be proved with the aid of an explicit construction for these graphs but not from the mere fact of their existence.

2. *Comparative Schematology*

We want to show that programs with coroutines are more powerful than programs with subroutines. We shall begin by explaining what we mean by these features. We shall then reduce our problem to one involving programs with neither coroutines nor subroutines but with one or more pushdowns. (This clarifies the problem by eliminating the notion of recursion.) We shall observe that programs with coroutines can efficiently simulate programs with any fixed number of pushdowns, while programs with one pushdown can efficiently simulate programs with subroutines. It will then suffice to show that programs with two or more pushdowns are more powerful than progrms with one pushdown, which will be done in the next section.

2.1 *Programs*

We shall consider subroutines, coroutines and pushdowns in the context of a simple programming language. We shall not attempt to define formally either the syntax or semantics of this language; we shall merely describe briefly its facilities. All of the assertions we shall make about the language should be obvious and their proofs, given appropriate formal definitions, should be routine.

Our language will have *global variables* which assume values either from an *uninterpreted domain* or from one of several *interpreted domains*. The interpreted domains will be taken to be free algebras with various combinations of generators. Traditional interpreted domains are *Boole* (two zeroadic generators), *Peano* (one zeroadic and one monadic generator), *Turing* (one zeroadic and two monadic generators) and *McCarthy* (one zeroadic and one dyadic generator). The *inputs* to a program (which may be either uninterpreted or interpreted) are to be found in certain designated global variables upon initiation, and the *outputs* are to be left in certain designated global variables upon termination.

Our language will have *blocks*, which may be nested, and *local variables* which may be either uninterpreted or interpreted and which have as their scope the smallest encompassing block.

Our language will have *assignment statements* which assign to one variable the current value of another variable or the result of applying a *function* to the current values of zero or more other variables. In addition to uninterpreted functions there will be interpreted functions: *constructors* corresponding to the generators of the free algebra and *selectors* corresponding to the arguments of the generators. (The selectors are, strictly speaking, partial functions.)

Our language will have *sequential, conditional* and *iterative statements*. The conditional and iterative statements will apply a *predicate* to the current values of zero or more variables. In addition to uninterpreted predicates there will be interpreted predicates corresponding to the generators of the free algebra. (Each element of a free algebra is in the range of a uniquely determined generator.)

2.2 *Subroutines and Coroutines*

Our language may be augmented with *subroutines* which may be invoked from the main program and which may invoke themselves or each other recursively. For simplicity, subroutines will not have parameters; all communication will take place through global variables (this allows the "call by copy" parameter mechanism -- see Snyder [8] -- to be simulated efficiently, however).

Note that nonrecursive subroutines are no more powerful than unaugmented programs (as can be shown by straightforwardly translating the former into the latter), but that recursive subroutines *are* more powerful than unaugmented programs (as can be shown by traditional computability considerations -- see Paterson and Hewitt [5]).

Our language may be augmented with *coroutines* which may be commenced and resumed from the main program and which, like the main program, can invoke subroutines. Coroutines, like subroutines, will have no parameters and will communicate through global variables.

Note that although we allow coroutines to invoke subroutines (which may in turn be recursive), we have no need for coroutines to commence and resume themselves or each other. This raises the question of whether recursive coroutines are even more powerful than the nonrecursive coroutines considered here. We conjecture that they are, but that the difference is small (quadratic rather than exponential).

2.3 Pushdowns

Our programming language may be augmented with one or more *pushdowns* which maintain values from the uninterpreted domain. (Note that the rich set of interpreted domains we have adopted makes it unnecessary to have pushdowns for these domains.) For each pushdown there will be *push* and *pop statements*.

It is easy to see that programs with one pushdown can efficiently simulate programs with subroutines, the pushdown being used to save and restore the values of uninterpreted local variables across invocations.

It is also not hard to show that programs with coroutines can efficiently simulate programs with any fixed number of pushdowns. In the scope of the declarations ***global Boole*** *flag* (***true*** means push and ***false*** means pop) and ***global uninterpreted*** *top*, the following coroutine simulates one pushdown.

```
co coroutine
    begin
        sub subroutine
            repeat
                begin
                    exit co;
                    if istrue flag

                        then begin
                                local uninterpreted save;
                                save ← top;
                                call sub;
                                top ← save
                             end
                        else exit sub
                end;
        call sub;
        call error
    end
```

After declaration and commencement of this routine, ***push*** *variable* can be translated into

```
flag ← true;
top ← variable;
resume co
```

and ***pop*** *variable* into

```
flag ← false;
resume co;
variable ← top.
```

The subroutine *error* is invoked if a pop is attempted when the pushdown is empty. By replicating this machinery, any fixed number of pushdowns may be simulated.

3. Pebbling with Auxiliary Pushdowns

We want to show that programs with two or more pushdowns are more powerful than programs with one pushdown. To do this we shall construct a functional that can be computed by a program with two pushdowns in polynomial time but for which programs with one pushdown require exponential time. This functional will be defined from a family of acyclic directed graphs. Two integral inputs will be used to select one graph from the family. The graph will have one vertex of in-degree zero (corresponding to an uninterpreted input); all its other vertices will have in-degree two (corresponding to the application of an uninterpreted function to two previously computed values). It will have one vertex of out-degree zero (corresponding to an uninterpreted output); all its other vertices will have out-degree one or more. In this way an appropriate family of graphs determines a functional which assigns to every dyadic uninterpreted function a function with two integral inputs, one uninterpreted input and one uninterpreted output.

3.1 The Pebble Game

To analyze the computation of functionals by programs, we shall use the "pebble game." (This is a one-player game and thus might better be termed a puzzle.) Consider an acyclic directed graph of the type described above. Consider a sequence of *moves* in which pebbles are put onto and taken off of the vertices of the graph according to the following rules.

Delete: A pebble may be taken off of a vertex at any moment.

Deposit: A pebble may be put onto a vertex whenever all the immediate predecessors of that vertex have pebbles on them.

A sequence of legal moves according to these rules will be called a *calculation* (< L. *calculus*, pebble). A calculation will be said to *pebble* a graph if it starts with no pebbles on any vertices and if every vertex has a pebble on it at some moment or another. The number of moves in a calculation will be called its *time*; the maximum number of pebbles on the graph at any moment will be called its *space*.

The pebble game for a graph models the computation of the corresponding functional by unaugmented programs, with pebbles representing variables assuming uninterpreted values and with deposits representing applications of uninterpreted functions.

For every graph, there is a certain minimum space required to pebble the graph.

Proposition 3.1.1: (See Hopcroft, Paul and Valiant [3].) *Any graph with N vertices can be pebbled in space $O(N/\log N)$.*

Proposition 3.1.2: (See Paul, Tarjan and Celoni [6].) *There exist graphs with N vertices for which pebbling requires space $\Omega(N/\log N)$.*

For any space at or above the minimum, there is a certain minimum time required to pebble the graph.

Proposition 3.1.3: (See Lengauer and Tarjan [4].) *Any graph with N vertices can be pebbled in space $S \geq 300\ N/\log_2 N$ and time $S \exp_2 \exp_2 O(N/S)$.*

Proposition 3.1.4: (See Lengauer and Tarjan [4].) *There exist graphs with N vertices for which pebbling in space $S \geq 300\ N/\log_2 N$ requires time $S \exp_2 \exp_2 \Omega(N/S)$.*

In the last proposition, the graphs depend on both N and S.

In order to model computations by programs with one or more pushdowns, we shall augment the pebble game with an equal number of *auxiliary pushdowns*, manipulated according to the following additional rules.

Push: If there is a pebble on a vertex, the name of that vertex may be pushed onto one of the pushdowns.

Pop: If the name of a vertex is at the top of a pushdown, a pebble may be put onto that vertex (even if its immediate predecessors do not all have pebbles) and the name popped off of the pushdowns.

The pebble game with auxiliary pushdowns models computations by programs with pushdowns in precisely the same way as the unaugmented pebble game models computations by unaugmented programs.

3.2 Upper Bounds

In this subsection we derive counterparts to Proposition 3.1.3 for pebbling with auxiliary pushdowns. The counterparts of Propositions 3.1.1 and 3.1.2 are trivial, for any graph with N vertices can be pebbled with one pushdown in space three. This is a consequence of the following stronger result.

Proposition 3.2.1: *Any graph with N vertices can be pebbled with one pushdown in space $S \geq 3$ and time $S \exp_2 O(N/S)$.*

Sketch of Proof: Use a recursive procedure that leaves pebbles on any $K = \lfloor S/3 \rfloor$ designated vertices of the graph. The procedure classifies the designated vertices according to whether they are among the last K or the first $N\text{-}K$ vertices in topologically sorted order. It calls itself recursively three times on the subgraph induced by the first $N\text{-}K$ vertices: twice to pebble the immediate predecessors of the last K vertices (after which the designated vertices among these can be pebbled) and once more to pebble the designated vertices among the first K. It uses the pushdown to save and restore pebbles across recursive calls. □

We shall see in the next subsection that Proposition 3.2.1 is the best possible.

Proposition 3.2.2: *Any graph with N vertices can be pebbled with two pushdowns in space $S \geq 3$ and time $O(N^2/S)$.*

Sketch of Proof: Use an iterative procedure that pebbles the vertices in batches of $K = \lfloor S/3 \rfloor$ in topologically sorted order. It uses its pushdowns to save the names of all vertices that have ever been pebbled, and makes one pass through these names to pebble the immediate predecessors of the vertices in each batch. □

We conjecture, but have not been able to prove, that Proposition 3.2.2 is the best possible. It is easy to see, however, that the lower bound $\Omega(N^2/S)$ would follow if $\Omega(N^2)$ could be proved for any fixed $S \geq 3$.

Proposition 3.2.3: *Any graph with N vertices can be pebbled with three pushdowns in space $S \geq 3$ and time $O(N \log (N/S))$.*

Sketch of Proof: Use a recursive procedure based on the "postman algorithm" of M. J. Fischer amd M. S. Paterson (see Pippenger [7], for example) to pebble the vertices in batches of $K = \lfloor S/3 \rfloor$ in topologically sorted order. □

We conjecture, but have not been able to prove, that Proposition 3.2.3 is the best possible. A simple counting argument can be used, however, to prove a lower bound of $\Omega(N \log N)$ for any fixed $S \geq 3$ and any fixed number of pushdowns. In particular, the exponential reduction in the sequence $S \exp_2 \exp_2 O(N/S)$, $S \exp_2 O(N/S)$, $O(N^2/S)$, $O(N \log (N/S))$ does not continue to $O(N \log \log (N/S))$ for four pushdowns.

The upper time bounds for pebbling with auxiliary pushdowns given above can be used to establish upper time bounds for computing the corresponding functionals by programs with pushdowns, but two additional issues must be addressed. First, we must consider the time needed to compute some representation of the graph to be pebbled. This will be dealt with in the next subsection, where it will be observed that the graphs we use can be computed in logarithmic space (suitably defined for our programs) and thus certainly in polynomial time. Second, we must consider the time needed to compute some representation of the calculation to be used for the graph. All of the procedures described above are quite straightforward and are easily implemented by our programs. In particular, the upper time bound for programs with coroutines can be obtained using Proposition 3.2.2, and it is easy to see that for fixed S this procedure can be implemented in logarithmic space and therefore also in polynomial time. Thus the polynomial upper time bound for

programs with coroutines applies both to the applications of uninterpreted functions and to the interpreted "housekeeping" operations.

3.3 Lower Bounds

In this subsection we derive a counterpart to Proposition 3.1.4 for pebbling with one auxiliary pushdown.

Proposition 3.3.1: *There exist graphs with N vertices for which pebbling with one pushdown in space $S \geq 3$ requires time $S \exp_2 \Omega(N/S)$.*

In this proposition, the graphs depend on both N and S.

The proof begins by considering graphs with in-degree d rather than in-degree 2, where d is a constant which will be determined later. Proposition 3.3.1 follows from Lemma 3.3.2 and Proposition 3.3.3.

Lemma 3.3.2: *A graph with N vertices and in-degree d can be transformed into a graph with dN vertices and in-degree 2 in such a way that a calculation pebbling the latter in space S and time T can be transformed into a calculation pebbling the former in space dS and time dT.*

The proof is straightforward.

Proposition 3.3.3: *There exist graphs with N vertices and in-degree d for which pebbling with one pushdown in space $S \geq 3$ requires time $S \exp_2 \Omega(N/S)$.*

The proof constructs the desired graphs by stacking bipartite graphs (provided by Lemma 3.3.4), considering the calculation at each level of the stack in turn and using induction on the number of levels in the stack.

Lemma 3.3.4: *There exist bipartite graphs with m^2 primary vertices, m^2 secondary vertices and degree d in which, if $S \leq m^2/72$, any S primary vertices are connected to at least $36S$ different secondary vertices.*

These graphs are constructed by taking the k-th power of the graphs described by Gabber and Galil [2] in their Theorem 2', where $k = \lceil \log_{(2-\sqrt{3})/4} 36 \rceil$. This yields $d = \exp_7 k$.

To analyze a stack of these graphs, consider the primary vertices V and the secondary vertices W at some level in the stack. If C is a calculation and U a set of vertices, let $\text{pebbles}_U(C)$ denote the sequence of subsets of U having pebbles at successive moments in C, and let $\text{deposits}_U(C)$ denote the sequence of vertices in U upon which pebbles are deposited (not popped) at successive moments in C.

If ρ is a sequence of vertices or sets of vertices, let the *diversity* of ρ be the number of different vertices involved. Let $\text{pack}_S(\rho)$ denote the maximum number of contiguous subsequences, each having diversity at least S, into which ρ can be parsed.

If a calculation C pebbles the graph, Lemma 3.3.4 implies

$$\text{pack}_{6S}(\text{pebbles}_W(C)) \geq 6\ \text{pack}_S(\text{deposits}_V(C)),$$

while Proposition 3.3.5 below implies

$$3\ \text{pack}_S(\text{deposits}_W(C)) \geq \text{pack}_{6S}(\text{pebbles}_W(C)),$$

so that

$$\text{pack}_S(\text{deposits}_W(C)) \geq 2\ \text{pack}_S(\text{deposits}_V(C)).$$

For a stack of height ℓ with $V_\ell = W_{\ell-1}, \ldots, V_2 = W_1$ we have

$$\text{pack}_S(\text{deposits}_{W_\ell}(C)) \geq 2^\ell\ \text{pack}_S(\text{deposits}_{V_1}(C)).$$

Since C takes time

$$T \geq S\ \text{pack}_S(\text{deposits}_{W_\ell}(C)),$$

while

$$\text{pack}_S(\text{deposits}_{V_1}(C)) \geq 1,$$

we obtain

$$T \geq S2^\ell,$$

which upon choosing $m = \lceil \sqrt{72S} \rceil$ and $\ell+1 = \lfloor N/m^2 \rfloor$ yields the desired lower bound.

Let $\text{init}_U(C)$ denote the sequence of vertex names in U on the pushdown just before the first move of C and let $\text{fin}_U(C)$ denote the corresponding sequence just after the last move.

Let $\text{cover}_S(\rho)$ denote the minimum number of contiguous subsequences, each of diversity at most S, into which ρ can be parsed.

Proposition 3.3.5: *For any calculation C and any set of vertices U,*

$$\text{pack}_{6S}(\text{pebbles}_U(C)) + \text{cover}_{2S}(\text{fin}_U(C))$$
$$\leq 3\ \text{pack}_S(\text{deposits}_U(C)) + \text{cover}_{2S}(\text{init}_U(C)).$$

This proposition, which is proved by straightforward combinatorial reasoning, is the heart of the proof. It says that during a calculation, many different vertices can have pebbles only if either there are many deposits or there is a substantial decrease in the "worth" of the sequence of vertex names on the pushdown. For a calculation C that pebbles the graph $\text{cover}_{2S}(\text{init}_U(C)) = 0$ and $\text{cover}_{2S}(\text{fin}_U(C)) \geq 0$, so

$$\text{pack}_{6S}(\text{pebbles}_U(C)) \leq 3\ \text{pack}_S(\text{deposits}_U(C)),$$

which completes the proof of the lower bound.

We observe that the construction due to Gabber and Galil used in Lemma 3.3.4, as well as the transformations used in Lemma 3.3.4, Proposition 3.3.3 and Lemma 3.3.2, can be carried out by our programs in logarithmic space. This completes the polynomial upper time bound for programs with coroutines. Proposition 3.3.1, however, yields an exponential lower time bound for programs with subroutines, even if only applications of uninterpreted function are counted and arbitrary interpreted computations are allowed without charge.

Acknowledgment

I am indebted to Hector Levesque, who asked how coroutines were more powerful than subroutines, and to Charles Rackoff, who pointed out the inadequacy of the traditional schematological answer.

References

[1] S. A. Cook, Characterizations of Pushdown Machines in Terms of Time-Bounded Computers, *J.ACM, 18* (1971) 4-18.

[2] O. Gabber and Z. Galil, Explicit Constructions of Linear Size Superconcentrators, 20th Ann. ACM Symp. on Foundations of Computer Science, 29-31 October 1979, San Juan, PR.

[3] J. E. Hopcroft, W. J. Paul and L. G. Valiant, On Time versus Space, *J.ACM, 24* (1977) 332-337.

[4] T. Lengauer and R. E. Tarjan, Upper and Lower Bounds on Time-Space Tradeoffs, 11th Ann. ACM Symp. on Theory of Computing, 30 April - 2 May 1979, Atlanta, GA.

[5] M. S. Paterson and C. E. Hewitt, Comparative Schematology, Proj. MAC Conf. on Concurrent Systems and Parallel Computation, 2-5 June 1970, Woods Hole, MA.

[6] W. J. Paul, R. E. Tarjan and J. R. Celoni, Space Bounds for a Game on Graphs, *Math. Sys. Theory, 10* (1977) 239-251.

[7] N. Pippenger, Fast Simulation of Combinational Logic Networks by Machines without Random-Access Storage, 15th Ann. Allerton Conf. on Communication, Control, and Computing, 28-30 September 1977, Monticello, IL.

[8] L. Snyder, An Analysis of Parameter Evaluation for Recursive Procedures, Ph.D. Thesis, Carnegie-Mellon University, 1973.

[9] H. R. Strong, High Level Languages of Maximum Power, 12th Ann. Symp. on Switching and Automata Theory, 13-15 October 1971, East Lansing, MI.