PROGRAM SCHEMATA AND INFORMATION FLOW
A STUDY OF SOME ASPECTS OF THE SCHEMA POWER
OF DATA STRUCTURES

TR 72 - 135

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University for the Degree of
Doctor of Philosophy

by
John Steven Brown

August 1972
PROGRAM SCHEMATA AND INFORMATION FLOW

A STUDY OF SOME ASPECTS OF THE SCHEMA POWER

OF DATA STRUCTURES

John Steven Brown, Ph.D.

Cornell University, 1972

The concept of a program schema, which represents the skeleton of a computer program, is important for its potential and actual applicability to code optimization, to understanding the behavior of programs using various control and data structure features, and to developing a theory of information flow as related to program form and power. In this study two storage devices, the tape and the queue, are considered, and their effect on the schema power hierarchy explored. The behavior and relative capability of each of the various types of tapes and queues is discussed in the light of the ways in which information is allowed to move, according to the inherent characteristics of the devices. Main results include the facts that addition of a single tape unit to the simplest schema class $P$ gives no additional power, and that the basic cyclic nature of queues and the persistence of data on tapes are important power determinants.

An attempt is made to bridge the gap between schema theory and actual programs not only by studying models of actual storage units but also by allowing an arbitrarily long (but finite) input stream. Such an extension generally upsets the
onion-like power inclusions of the finite input hierarchy because of the calculations possible on the implicitly-input arbitrary integer, which is the number of input values. The information flow characteristics of the data structures associated with a particular schema class may allow such an arbitrary input to be viewed exactly once, exactly twice, or in general \( m \) times, and such distinctions give rise to differences in computational power.

Requirements insuring that the interconnection of schemata form a schema with the union of the capabilities of the subschemata are considered. The hierarchy of schemata without the identity assignment is also studied, and it is demonstrated that in the simplest class \( P \) and the universal class \( P_{\text{AE}} \) no loss of power results from prohibiting identity assignments. Discussion of the as-yet-unsolved problem of the power of the class \( P_{(2b,0)} \) and its associated derivatives is presented. A summary of prospects for developing program schema research in several different potentially practical directions concludes the work.
BIOGRAPHICAL SKETCH

John Steven Brown was born in Hendersonville, North Carolina, on November 19, 1946, the son of Mr. and Mrs. John Lowrance Brown. He attended public schools in that city and was graduated from Davidson College in 1968 with the degree of Bachelor of Science, summa cum laude, having majored in physics. In September of that year he entered the graduate school at Cornell University to pursue studies in computer science, and was awarded the degree of Master of Science in June of 1970. He is a member of Phi Beta Kappa, Omicron Delta Kappa, Phi Kappa Phi, and the Association for Computing Machinery. The author is married to the former Geraldine Rustin.
ACKNOWLEDGEMENTS

The author is particularly grateful to his advisor, Dr. David Gries, for his continued support, patience, and encouragement, and to Dr. John Hopcroft and Dr. Richard Conway, the other members of his special committee. Many stimulating discussions and much encouragement were also given by Tom Szymanski, Jean-Pierre Lévy, and Paul Reilly, to whom the author owes many thanks. Mrs. Maria Laptewicz did a speedy and careful job of typing the manuscript.

The National Science Foundation has supported the bulk of this research, both through the author's four-year term of an NSF Traineeship and through its grant GJ-28176.

Finally, and most importantly, the author thanks his wife for her love, support, and confidence during the sometimes trying period of this work. To her, thanks and ILU ....
TABLE OF CONTENTS

1. INTRODUCTION
   1.1 Survey of the History of Schemata 1
   1.2 Four Aims of Schema Research 3
   1.3 Concept of Information Flow 6
   1.4 Overview of Notation and Conventions 11

2. QUEUE DATA STRUCTURES
   2.1 Definition of Queues in Schemata 15
   2.2 Queues without Chips or Markers 19
   2.3 Single Queues with Chips 22
   2.4 Multiple Queues with Chips 35
   2.5 Emptiness-Testable Queues 38
   2.6 Example of the Application of Queues to Matrix Multiplication 49

3. TAPE DATA STRUCTURES
   3.1 Definition of Tapes in Schemata 56
   3.2 One-Tape Classes with One-Way and Two-Way Reading Capability 58
   3.3 Two-Tape Schemata 72
   3.4 Tape Schemata with the Write-after-Read Capability 75
CHAPTER 1. INTRODUCTION

1.1 Survey of the History of Schemata

Problems to which computers are applied can be considered as functions from a set of inputs in a domain \( D \) to a set of outputs in a domain \( D' \), but most frequently such problems are studied in the light of algorithms which are specified for their solution. The notion of an algorithm in this context is somewhat imprecise since there is no restriction on the language in which it is to be stated. Even if a "sufficiently powerful" (in the sense of being able to express all partial recursive functions) language \( L \) is considered, the statement of an algorithm for a particular problem is still in general nonunique because the algorithm may call for irrelevant computations, may specify unnecessary ordering constraints, may require duplicate calculations, etc. Furthermore, there may exist algorithms \( A \) and \( A' \), both expressed in \( L \), which use different features of \( L \); for example, if \( L \) were PL/I, \( A \) might be expressed in terms of array variables, while \( A' \) might accomplish the same result as \( A \), but using controlled storage allocation. Studies in the theory of computation have established that in general, equivalence and related questions (for \( A \) and \( A' \) over an \( L \) of the type above) are undecidable, and even for restricted situations in which such questions are theoretically decidable, the decision process is extremely complex and costly.

In the 1950's, consequently, as the study of programming languages became popular, researchers noted that problems such
as mechanical program optimization are concerned principally with the application of equivalence preserving transformations to given programs. Questions of equivalence therefore became of practical importance. This motivation, together with factors such as McCarthy's influence on the development of a mathematical science of programming, led to the concept of a program schema. A schema is the expression of the skeleton of a normal program, with the dual intent of simplifying structure enough to make certain questions answerable and of removing enough material from real programs to allow the roles of certain features to become understandable. More than one type of skeleton exists, of course, though the principal type investigated has been that obtained by leaving all or certain classes of predicates and data transformations expressed as unspecified functions in the static program representation. Another formulation that has been studied is a partially uninterpreted specification of the trace of a program during execution.

During the first years of schema investigation (a good summary of which can be found in Ershov's survey [Ersh71]), Russian researchers carried out the bulk of the work. Ianov and others investigated a wide variety of schema types and program histories with a view to recognizing equivalences and making suitable transformations among comparable programs to produce equivalence. The 1960's were largely fallow years in schema research, but the work of M. S. Paterson [Pat67], published as his Ph.D. thesis in 1967, restimulated active work in that area. Since that time, schema research has
branched out along several paths, such as consideration of parallelism from a schema viewpoint (Karp and Miller [Karp69], and others), study of effective functionals and schemata (Strong [Str71b]), investigations of "power" relationships among schema types (Strong [Str71a], Paterson and Hewitt [Pat70], Constable and Gries [Con72], Chandra and Manna [Chan71], Brown, Gries, and Szymanski [Bro72]), and work continuing the earlier equivalence studies.

1.2 Four Aims of Schema Research

From the earliest investigations up to the present, one of the goals of the schema approach to the study of programming languages has been to develop a more systematic and more effective theory of program optimization. Ideally, of course, a translator would be able to identify the transformations T1, T2, and T3 in Figure 1.1 and thereby be able to create, by considering the abstracted essence of a program, an equivalent optimal program. For the restricted case of straightline code under some limitations, Aho and Ullman [Aho72] have formulated the optimization problem (with respect to "removal of useless statements and variables, identification of two computations producing the same value, renaming variables, and flipping of adjacent statements") in this way, but the general optimization scheme of this style still seems to be some distance away. True optimization
Figure 1.1 — Anticipated role of schemata in code optimization

requires that every equivalent program be abstractable into a
schema form which is recognizably equivalent to every other
such schema form, and furthermore, that an effective cost
function be defined by which optimality can be judged; amelio-
ration requires, of course, only the cost function. At present,
research has not shown a method of this form that can guarantee
completeness in the program representations it considers (that
is, absolute optimality as opposed to optimality relative to
a subset of the programs which compute a particular function),
and furthermore, little work has been done in defining and
working with cost functions. A complication of a cost function
$CF$ is that if $CF$ is defined over schemata, the ordering it
imposes may not be preserved at the program level, whereas
if $CF$ is defined over programs, it is less convenient (more
expensive) to apply. In addition, $CF$ is dependent on (and
conversely can influence) hardware design of "real" machines
in the same way that available primitives (excluding, say,
microprogramming from consideration) affect the way in which
a program is schematized. In spite of these difficulties,
the goal of an orderly theory of optimization (and its con-
sequent of improved optimizing compilers) seems to be an attain-
able one, though much more work will be required.
A second aim of schema investigations is to develop an understanding of how and why a particular feature — data structure, storage device, control structure, etc. — "works" or adds capability. A companion objective is to separate the influence of control structures from that of data structures by means of the schema abstraction; experience has shown that often a tradeoff between added control features (such as markers or label variables) and weakened data structure types can be formulated, and such results are again important in code optimization. Examples of understanding achieved through schemata studies include the close relationship of recursion and pushdown stores (perhaps intuitively expected but systematically examined by Constable and Gries [Con72] and by Brown, Gries, and Szymanski [Bro72]), and the power of detectable markers that lie outside the range and domain of basic functions. This last topic may have its principal import in theoretical studies of schema class power distinctions, but it should also be noted that such markers have application to the real-world debugging problem of finding an inexpensive way to detect fetches from uninitialized variables. Additional instances of such hows and whys are developed in some of the remainder of this thesis.

Two other potential ends of schema research which have not been so widely recognized are additional clarification of the relationships between the abstract machines of investigators in the theory of computation and "real" machines, and the development of the schema approach to the point where it
can be used to suggest and evaluate new data structures or control structures appropriate to specific problems. The evaluation is carried out on the basis both of algorithmic feasibility and of the translatability of schemata using the structures into equivalent schemata using other data structures. Much work remains to be done on both of these topics, but some results in each area have been obtained and are outlined below; for example, a particular tape-augmented schema class has been shown equivalent in structure to stack automata (see Hopcroft and Ullman [Hop69], Chapter 13), and considerable work has been done relating to queues as program data structures.

1.3 Concept of Information

A thread that runs through schema research and which is important for understanding the basic nature of an algorithm is the underlying theme of information flow. Basically, an algorithm specifies a transformation — usually by means of a sequence (more or less ordered) of sub-transformations — on the information contained in given input in order to produce certain output. The mapping so specified may be conditioned on the input (e.g., defined only for certain inputs, different for different classes of inputs, etc.), but for a particular input or class of inputs the algorithm can be thought of as specifying a well-defined sequence (up to a restricted permutation) of transformations which must be calculated to accomplish the function defined by the algorithm. If the algorithm is expressed in terms of a specific set of data structures
(rather than in terms of generic commands such as "save" or "fetch"), it further designates a sequence of data structure references. One may say that these sequences characterize the essence of the algorithm, though it should be noted that they by no means uniquely characterize the problem which the algorithm solves (i.e., the function it computes).

Much of the "comparative schematology" research has been directed toward defining classes of functions computable by a definite machine configuration and the relationships among those configurations, but the results could be interpreted as showing how the constraints of a structure on information flow determine the class of functions computable by that structure. To make the concept clearer (and at the same time cite a real world situation to which this schema research applies) one can consider a railroad switching yard (Tarjan [Tar72] has considered such a device in a related context) in which two-way self-propelled cars enter from one side of the yard and pass through and out the other side under the control of a signal system. Within the yard are loading docks which can transfer goods into or out of cars, and unlimited empty cars and storage for empty cars. There is also a supply of white cabooses (which never enter or leave the yard) and a sequence of track-side detectors which can either read the sign that each car carries denoting its contents or recognize the cabooses, and which can cause switches to change position.

Such a yard clearly represents a schema language, where the operation of the loading docks and the behavior of the
trackside detectors correspond to uninterpreted functions and predicates, respectively. Figure 1.2 shows a possible yard configuration to calculate the Leaftest(P,L,R,x) functional of Paterson and Hewitt [Pat70], and Figure 1.3 shows track configurations corresponding to different data structures of ordinary schematology. The layout of Figure 1.2 is noticeably different from a yard to compute Leaftest using push-down stores and caboose markers; the algorithms are different because different data structures are used in their conception. Signals, which require fairly complicated interconnections, are used in Figure 1.2 chiefly to prevent collisions, though they can in general be used for more complex control; any car facing a green signal at time t can move then (for simplicity and to restrict the considerations to sequential schemata, only one car is permitted to move at a time, and all moves are accomplished within one time unit).
Legend:

\[ \begin{align*}
\begin{array}{c}
\text{x} \\
\infty \\
\infty
\end{array}
\end{align*} \quad \text{self-propelled car containing x}
\]

\[ \triangle \quad \text{trackside detector} \]

\[ \begin{align*}
\begin{array}{c}
\text{L(x)}
\end{array}
\end{align*} \quad \text{loading dock — unloads x, then loads L(x)}
\]

\[ \square \quad \text{signal flip flop} \]

\[ \underline{S} \quad \text{signal set to state S} \]

Figure 1.2 — Algorithm to compute Leafest
Figure 1.3 – Track configurations for various data structures

(More complicated devices such as arrays can also be represented; for example, arrays can be realized by a turntable whose stepping is controlled by a count (of cars) and whose behavior is regulated by a complicated connection of signals.)

Pursuing the analogy further is perhaps unfruitful, especially since certain computation concepts, such as a rereadable tape, seem to have no analogue in railroading, and since other concepts, such as nondestructive readout of the value of a variable, are awkward to express. The point of view of information moving through some kind of a graph with nodes having special properties is valid, but suffers chiefly from inferior notation. Similarly, the layout of railroad yards where complex car reordering is necessary (based on the content
or identity of the cars) can be derived from schema results.

The material in the following sections of this thesis is regarded as, in some sense, data for an as yet undeveloped systematic theory of information flow in programs. Several configurations are studied, most notably those containing queues, tapes, and an unknown quantity of input (finite but unbounded length), and in each case an effort is made to relate the new structure to other structures in terms of the size of the class of functions it can compute.

1.4 **Overview of Notation and Conventions**

The research of this thesis follows closely the spirit, notation, and approach of earlier papers by Constable and Gries [Con72], and Brown, Gries, and Szymanski [Bro72], in which an attempt was made to relate schema research closely to real programming languages. Schemata are expressed in terms of a simple high-level language which employs in addition to specified control features, uninterpreted basic functions and predicates; the features incorporated in the language are varied in order to examine different schema classes (or machines), but in all cases only serial schemata (isomorphic to flow-chart schemata) are studied. A single interpreted function, the identity function, is present in all schema classes (the effect of dropping the identity function is briefly discussed in Chapter 6), and strictly constrained markers (outside the range and domain of all basic functions), either in unbounded or limited supply, are sometimes used to augment
the control structure. Characteristic data structures (some with features such as bottom testability which impinge on control structure) such as arrays, pushdown stores, queues, etc. are incorporated singly or in homogeneous and heterogeneous groups to create machine classes. In keeping with the goal of isolating control structures and data structures from each other and from basic predicate/function behavior, domain elements are kept strictly isolated from control elements such as markers and subscripts; not all researchers have adhered carefully to this convention, but it seems most reasonable to avoid confusion by keeping the two types of components separate.

Considerable familiarity with the two above references is assumed in the remainder of this thesis. In particular, the basic definitions of the various schema classes, the definition of the inclusion relation between classes, and the concepts of locators and p-simulators are frequently used; a brief summary of the relevant prerequisite material is given in Section 1 of [Bro72].

Besides the general philosophy and organization of approach, perhaps the most important heritage from earlier work is the concept of a universal schema class, of which a representative form is $P_{\text{AE}}$ (basic program schemata augmented with arrays and subscript equality tests). It is hypothesized that this class, together with the effectively equivalent classes $P_{(3b,0)}$, $P_{AM'}$ etc., has sufficient power to mimic any domain independent control structure or data structure for serial execution. Note that the concept of universality does not deal
with domain elements and their contents, since the domain $D$ of a computation is assumed to be completely arbitrary and unknown. Hence, although a schema class augmented with a predicate to test the equality of two domain elements can indeed compute functions that the "universal" classes above cannot calculate, the two classes are really incomparable because one assumes more information about the domain than does the other.

Throughout the work reported here direct proofs by simulation appear, and a main feature of many such proofs is the encoding of data structures by a method termed interleaved Gödelization. Basically, the technique is the following:

Let the data structures being encoded into structure $ENC$ be $A, B, \ldots, Z$, with ordered elements $a_1, a_2, \ldots, a_{|A|}$, $z_1, \ldots, z_{|Z|}$, where, as usual the notation $|A|$ means the number of elements in $A$ (since data structures are nearly always unbounded in potential size, $|A|$ is considered to be the net number of elements which have been inserted into $A$ since the schema of which $A$ is a part began operation).

Distinct primes $p_1, p_2, \ldots, p_z$ are associated, respectively, with $A, B, \ldots, Z$. Generally, the smallest of these primes is chosen to be larger than the number of structures being encoded, for ease of checking certain conditions in some of the algorithms.
Then the elements are stored within $\text{ENC}$ so that they are accessed in the following order:

\[
\begin{array}{cccccccc}
a_1 & b_1 & \cdots & z_1 & a_2 & b_2 & \cdots & z_2 & \cdots & a_n & b & B & c_1 & \cdots & z_n & a_{n+1} & \Omega & c_{n+1} & \cdots & \Omega
\end{array}
\]

Padding symbols $\Omega$ of any value are inserted as place holders if, as shown, $|B| < |A|$, etc. Furthermore, padding symbols $\Omega$ are appended after all data elements until $|\text{ENC}| = p_1 |A| p_2 |B| \cdots p_z |Z|$.

The method of storage may also be in reversed order from that shown, but the essential plan is unchanged.

Another often-used device is retention of a fixed amount of information by replicating copies of a schema. This technique is analogous to remembering items using states of the finite control in automata models. Basically, if it is desired to keep track of condition $x$ during execution of a schema $S$, two copies $S'$ and $S''$ of $S$ are employed, one of which corresponds to $x$ and the other to $\text{not } x$. In $S'$ all predicates depending on $x$ are known to be true and in $S''$ they are known to be false. In $S'$ any action (at statement $j$) which causes $\text{not } x$ to hold is changed to cause control to move to the successor of the twin of statement $j$ in $S''$. Corresponding changes are made in $S''$ for actions which cause $x$ to hold. The extension of the technique to several conditions, each of which can assume several values, should be clear.
CHAPTER 2. QUEUE DATA STRUCTURES

2.1 Definition of Queues in Schemata

Earlier schema studies have investigated structures such as pushdown stores, in which the information flow could be thought of as LIFO (last in, first out), and arrays, in which the information flow can be considered as random (without considering the pattern of subscript generation). An information flow point of view then leads naturally to consideration of queue-like storage devices, in which the flow is FIFO (first in, first out) or some combination of LIFO and FIFO. The possible queue types can be represented by the following diagrams, where arrows represent the possible data flows at each end of the queue:

Type A  \[ \rightarrow \] \[ \leftarrow \]  \rightarrow 
Type B  \[ \rightarrow \] \[ \leftarrow \]  \leftarrow 
Type C  \[ \rightarrow \] \[ \leftarrow \]  \rightarrow 
Type D  \[ \leftarrow \] \[ \rightarrow \]  \leftarrow 

Knuth [Knu68] has referred to these data structures as, respectively, input restricted deque, output restricted deque, queue, and deque. A stack can also be represented in this notation:

Type E  \[ \leftarrow \] \[ \rightarrow \] 

All other combinations of data flows in queue-like devices are either uninteresting (because of restrictions which would not allow both insertion and removal of data) or functionally
equivalent to one of the above types.

FIFO and LIFO when used alone are clear in their implications to data flow. When used together, an additional time parameter must be introduced to clarify what is meant, and it is this time parameter that is characterized by the four types of queues. In a single queue the possible flows might be described as

FIFO ahead of LIFO, or B-type: →
FIFO behind LIFO, or A-type: ←
FIFO both behind and ahead of LIFO, or D-type: ↔

with the understanding that the FIFO and LIFO data streams are, in themselves, unseparated. The full implications of these combinations of flow have not yet been established.

Certain conventions and accessing methods can be defined in order to permit queues of these kinds to be used with program schemata:

a) **Accessing functions** — There are functions PUSHLEFT (queueName,variable), PUSHRIGHT(queueName,variable), POPLEFT(queueName,variable), and POPRIGHT(queueName, variable), which push or pop a value from the left or right end of a queue, according to their names. Not all functions are valid with every queue type, and execution of an undefined operation for a particular type is treated as a null statement, as is a pop operation performed on an empty queue.
b) **Initialization** — Queues are empty when execution of a scheme is begun.

c) **Augmentation with chips** — Queue schemata may be augmented with one or more chips, which are distinguished from markers in that there can be only one instance of each chip (whereas there may exist unlimited copies of a marker). Chips have been introduced in the paper of Brown, Gries, and Szymanski [Bro72], but a brief summary of their behavior will be given here:

1) A variable \( v \) can be tested to see whether it contains the chip \( C \) by the statement

\[
\text{IF } v = C \text{ THEN } S_1 \text{ ELSE } S_2
\]

where \( S_1 \) and \( S_2 \) are statements.

2) A chip is shifted among simple variables during execution so that it always resides in the variable to which it was last assigned (other variables receive the value \( \Omega \) as the chip is moved away), but if the chip is placed into a data structure (queue in this case) it remains there until explicitly fetched by an accessing primitive for the structure.

As an example of chip behavior, suppose a schema has only one chip \( C \). \( C \) may be referred to by name, as in \( x + C \), in which case \( \text{content}(x) \) becomes \( C \), as long as the chip is not currently in some data
structure. If the next executed statement is \( w \rightarrow C \), or \( w \rightarrow x \), then content\((w)\) becomes \( C \) and content\((x)\) becomes \( \Omega \). If a statement like \( \text{PUSHLEFT}(Q, C) \) or \( \text{PUSHLEFT}(Q, w) \) is then executed, the chip is placed in the queue, where it remains until explicitly fetched by the proper \( \text{POPRIGHT}(Q, z) \) (say) statement, and content\((w)\) becomes \( \Omega \) immediately after the push. While \( C \) is in the queue it is not available for assignment to simple variables or for insertion into the queue, so during that time execution of the statement \( x \rightarrow C \) would result in content\((x)\) becoming \( \Omega \), and execution of \( \text{PUSHLEFT}(Q, C) \) would result in \( \Omega \) being pushed onto the queue.

d) **Augmentation with emptiness test** - Queue schemata may be permitted to use the statement

\[
\text{IF EMPTYQUEUE}(\text{queue}name) \ \text{THEN} \ S1 \ \text{ELSE} \ S2
\]

which causes statement \( S1 \) to be executed if the queue referenced contains no elements, and \( S2 \) to be executed otherwise.

A general schema class \( P_x \) augmented with \( k \) queues is denoted by a name of the following types, where the first notation represents any of the four queue designs: \( P_x \ kQ_A \), \( P_x \ kQ_B \), \( P_x \ kQ_C \), \( P_x \ kQ_D \). Chip-augmented queue schemata are designated as \( P_x \ kQ_Z(n) \), where \( n \) is the number of chips and \( Z \) is \( A, B, C, D, \) or \( \Lambda \). The addition of the emptiness
test is denoted by $P_x kQ_z(n)b$.

2.2 *Queues without Chips or Markers*

The following three lemmas, stated without proof because of their simplicity, characterize some of the basic relationships among the types of queues.

\[(2.1) \text{Lemma: A particular function } f \text{ can be computed in } P_{kQ}(n) \text{ if } f \text{ can be computed in } P_{kQ_C}(n).\]

\[(2.2) \text{Lemma: } P_{mQ_D}(n) \geq P_{mQ}(n), \text{ for all } m \in \{1, 2, 3, \ldots\} \text{ and } n \in \{0, 1, 2, \ldots\}.\]

\[(2.3) \text{Lemma: } P_{mQ_B}(n) \geq P_{mQ_C}(n), P_{mQ_A}(n) \geq P_{mQ_C}(n), \text{ for all } m \text{ and } n \text{ as above.}\]

As mentioned earlier, from an information flow point of view, the principal difference between pushdown stores and queues is that the flow in the former is LIFO, whereas in the latter it is FIFO. It is perhaps surprising to find that in the absence of markers or chips the FIFO discipline allows computation of a larger class of functions than does the LIFO. This result is shown by the ability of a simple queue to compute the Leafiest functional (which Brown, Gries, and Szymanski ([Bro72], Lemma 3.3) showed noncomputable in $P_{(n,0)}$), and the adequacy of a C-type queue for the construction of a $p$-simulator for a schema in $P_{(n,0)}$.

\[(2.4) \text{Lemma: Leafiest}(P, L, R, x) \text{ can be computed in } P_{1Q}(0).\]

**Proof** The following construction demonstrates how to
compute the functional using only operations permitted to C-type queues:

\[(x): \text{PUSHLEFT}(Q,x); v \leftarrow x; \text{UNTIL } P(v) \text{ DO}
\]
\[\begin{array}{l}
\text{BEGIN POPRIGHT}(Q,v); \\
\text{PUSHLEFT}(Q,L(v)); \\
\text{PUSHLEFT}(Q,R(v)) \\
\text{END;}
\end{array}
\]
\[\text{HALT}(x);\]

This schema halts, displaying \(x\), iff Leafest is satisfied. Q.E.D.

The operation of the schema to compute Leafest is straightforward: a value \(v\) is shifted out of the queue and tested under the predicate \(P\); if \(P(v)\) yields true, the schema halts. Otherwise, the values \(L(v)\) and \(R(v)\) are pushed onto the queue and the process is continued.

(2.5) **Theorem:** \(P_{1Q_C}(0) > P(n,0)\).

**Proof** To construct a schema \(S'\) in \(P_{1Q_C}(0)\) to simulate \(S\) in \(P(n,0)\), the first step is to construct the locator for \(S\). The existence and form of such a locator, which lies in the class \(P < P_{1Q_C}(0)\), are given in [Bro72]. At the labels BEGINi, where the locator transfers control when a predicate \(p\) and values RT and RF have been found such

\[1\text{Where convenient, certain constructions such as UNTIL and WHILE clauses, which are, strictly speaking, not part of the schema language, are used for brevity and clarity. In each case, simulation of such constructions using IF statements is straightforward (Section 2 of [Con72] gives some examples of these extensions).} \]
that \( p(\text{RT}) = \text{true} \) and \( p(\text{RF}) = \text{false} \), new copies of \( S \) are attached in which all \( pds \) pushes and pops have been replaced by \( \text{BEGIN}...\text{END} \) blocks which simulate the actions using the queue \( Q \). The contents of the \( pds \)'s of \( S \) are stored in order, one after the other, separated by markers, in \( Q \):

\[
\rightarrow \text{pds}_1 \text{marker} \text{pds}_2 \text{marker} \cdots \text{pds}_n \text{marker} \rightarrow
\]

The markers are distinguished from ordinary elements by allotting two queue cells to each value stored. The first cell \( c \) of each pair is \( \text{RF} \) iff the second cell contains a member of one of the \( pds \)'s; otherwise, \( c \) is \( \text{RT} \) (and the second cell's content is irrelevant). Pushes and pops of \( \text{pds}_j \) are accomplished by circularly shifting the queue past \( n - j + 1 \) markers, shifting to the next marker, performing the addition or deletion (deletions are achieved by using a buffer), and continuing the circular shifts until the canonical configuration is restored.

The strict inclusion follows at once from Lemma 2.4 and from Lemma 3.3 of [Bro72].

Q.E.D.

Outside of the above result, not much has yet been ascertained about the classes \( P_{kQZ}(0) \), but it is conjectured that \( P_{1QZ}(0) \equiv P_{nQZ}(0) \), for \( Z = A, B, C, D \).

The configuration of a bare \( C \)-type queue is like that of an expandable (in terms of storage capacity) recirculating
register (such a register has either an associated clock or a
timing mark to separate its end from its beginning). As might
be expected from the previous result about FIFO superiority
to LIFO discipline, it is easy to show that adding integer
arithmetic to the class $P_{1Q(0)}$ gives universality:

(2.6) **Lemma:** $P_{1Q(0)+\mathbb{N}}$ is universal.

**Proof** The result follows at once from Theorem 10.10 of
[Con72], which states that $P_{(1,0)+\mathbb{N}}$ is universal. The queue
is used as the pds by keeping an extra integer variable which
records the number of elements currently on the pds. Circular
shifting of the queue using that count then allows proper
simulation of the pushing and popping of the pds.

Q.E.D.

2.3 **Single Queues with Chips**

It has been noted that queue data structures correspond
to a FIFO, or potentially circular, information flow, and it
is intuitively expected that addition of the capability to mark
one or more places in the data would give more power. The most
restrictive place-marking endowment is the addition of a small
number of chips, and it is surprising that only two chips
(corresponding to dividing the data into three parts) in a
C-type queue are sufficient to give universal power. One chip,
when added to a D-type queue, creates a configuration that is
equivalent to the elusive class $P_{(2b,0)}$.\(^2\) As in the no-chip

\(^2\)It was hoped that this result would be helpful in solving the
problem of whether $P_{(2b,0)}$ is a universal class, but so far
that question is still unresolved.
case, there are many open questions for chip-augmented queue schemata, principally because of the rather large number of possibilities for schema classes and comparisons.

The proof that \( P_{1Q_D}(1) \) is equivalent to \( P_{(2b,0)} \) is established, as are many of the proofs that follow, by a direct simulation. This result can be best understood by recalling that a D-type queue appears as \( \leftrightarrow \). The simulation of two pds's operates by inserting the chip \( C \) into the queue \( Q \) and using the portion of \( Q \) to the left of \( C \) as the first pds \( PD_1 \) and the portion to the right of \( C \) as \( PD_2: \leftrightarrow PD_1 C PD_2 \leftrightarrow \).

\[(2.7) \textbf{Theorem: } P_{1Q_D}(1) \equiv P_{(2b,0)}.\]

\textbf{Proof} Let \( S \) be a schema in \( P_{(2b,0)} \). An equivalent schema \( S' \) in \( P_{1Q_D}(1) \) is constructed from a copy of \( S \) by making the following changes:

1) The queue is initialized by inserting the statement \( \text{PUSHLEFT}(Q,C) \) at the beginning of \( S' \).
2) \( \text{PUSHLEFT}(Q,v) \) is substituted for every statement \( \text{PUSH}(PD_1,v) \) of \( S \).
3) \( \text{PUSHRIGHT}(Q,v) \) replaces each instruction \( \text{PUSH}(PD_2,v) \) of \( S \).
4) Every \( \text{POP}(PD_1,v) \) of \( S \) is replaced by the block
   \[
   \begin{align*}
   \text{BEGIN} & \text{POPLEFT}(Q,Z); \text{IF } Z = C \text{ THEN } \text{PUSHLEFT}(Q,Z) \\
   & \text{ELSE } v \leftarrow Z \\
   \text{END}
   \end{align*}
   \]
5) Every \( \text{POP}(PD_2,v) \) is changed to the similar group
   \[
   \begin{align*}
   \text{BEGIN} & \text{POPRIGHT}(Q,Z); \text{IF } Z = C \text{ THEN } \text{PUSHRIGHT}(Q,Z) \\
   & \text{ELSE } v \leftarrow Z \\
   \text{END}
   \end{align*}
   \]
6) Every statement IF EMPTYPDS(PD1) THEN S1 ELSE S2
is replaced by the compound statement
BEGIN POPLEFT(Q,Z);
   IF Z = C THEN BEGIN PUSHLEFT(Q,Z); S1 END;
   ELSE BEGIN PUSHLEFT(Q,Z); S2 END;
END
where Z is a new variable not used in S. Because
of the behavior of the chip (only one copy exists)
the pushes cannot be moved back before the test.

7) Each instruction IF EMPTYPDS(PD2) THEN S1 ELSE S2
is replaced by a similar block except that all queue
suffixes are RIGHT instead of LEFT.

A schema $S'$ constructed by the above method clearly simulates
a given S in the class $P_{(2b,0)}$, thus establishing that
$P_{LOD}(1) \geq P_{(2b,0)}$.

To establish the inclusion on the other direction a similar
simulation will be employed. If S is a schema in $P_{LOD}(1)$
which uses $k$ simple variables $v_1, v_2, ..., v_k$ then the
simulating schema $S'$ is constructed by making $k+2$ copies
of S and linking them together. The copies are named
$S^0, S^1, ..., S^{k+1}$, and the statements of each copy are all
labeled as $s_{ij}$, where $j$ is the copy number and $i$ is the
index of the statement in S; all GO TO's, etc. are changed
so that each copy is equivalent to S with renaming. Execution
will be in copy $S^0$ if the chip has not been used, in
copy $S^{k+1}$ if the chip is in the queue Q, and in copy
$S^j (j \in \{1, ..., k\})$ if the chip is in variable $v_j$. 
The copies are modified for communication according to the following plan:

1) If the marker is not in the queue, the contents of the queue are kept left end "up" in pds PD1. PUSHLEFT and POPLEFT are then simple pds manipulations; POPRIGHT and PUSHRIGHT are handled by pouring the contents of PD1 into PD2 and using analogous pds functions.

2) If the marker is in Q, it stands logically between the bottom element of PD1 (which contains the left-hand portion of Q left end "up") and the bottom element of PD2 (which contains the right-hand portion of Q right end "up"). Symbolically,

\[
Q: \quad \longleftrightarrow q_1 \cdots q_m \quad C \quad q_{m+2} \cdots q_n \quad \longleftrightarrow
\]

is represented by

\[
\begin{array}{ccc}
\text{PD1} & & \text{PD2} \\
q_1 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{array}
\]

The pushes and pops are then handled in a straightforward way, with notice being taken when C is popped (one of the pds's is emptied).

3) The different copies of S are employed to detect when the chip is pushed onto the queue.
The rather involved details of the simulation transformations are as follows:

1) In copy $S^0$

Every \textsc{pushleft}(Q,v) is replaced by \textsc{push}(PD1,v).

Every \textsc{popleft}(Q,v) is replaced by \textsc{pop}(PD1,v).

Every \textsc{pushright}(Q,v) is replaced by (using a new variable $Z$)

\begin{verbatim}
BEGIN UNTIL EMPTYPDS(PD1) DO 
  BEGIN POP(PD1,Z); PUSH(PD2,Z) END;
  L: PUSH(PD1,v);
  UNTIL EMPTYPDS(PD2) DO 
  BEGIN POP(PD2,Z); PUSH(PD1,Z) END;
END.
\end{verbatim}

Every \textsc{popright}(Q,v) is replaced by the same block of code except that statement $L$ is replaced by \textsc{pop}(PD2,v).

Every statement $\lambda^0_i$: IF $v_j = C$ THEN $S1$ ELSE $S2$ is replaced by $\lambda^0_i$: $S2$.

Every statement $\lambda^0_i$: $v_j$ + $C$ is replaced by the block $\lambda^0_i$: BEGIN $v_j$ + $\Omega$; GO TO $\lambda^1_{i+1}$ END (C behaves as the value $\Omega$ in computations expecting a value from the domain D).

2) In copy $S^j$ ($j \in \{1, \ldots, k\}$)

Every \textsc{pushleft}, \textsc{popleft}, \textsc{pushright}, and \textsc{popright} for which the target variable is not $v_j$ is transformed as in $S^0$.

Every statement $\lambda^j_i$: \textsc{pushleft}(Q,v_j) is replaced by the group $\lambda^j_i$: BEGIN \textsc{push}(PD1,v_j); $v_j$ + $\Omega$;

GO TO $\lambda^{k+1}_{i+1}$ END and similarly (using the pouring
as in copy $S^0$) for instances of $\text{PUSHRIGHT}(Q,v_j)$.

Every statement $\ell_i^j$: $\text{POPLEFT}(Q,v_j)$ is replaced by
the unit $\ell_i^j$: BEGIN $\text{POP}(PD_1,v_j)$; GO TO $\ell_i^{0+1}$ END
and similarly for $\text{POPRIGHT}(Q,v_j)$ statements.

Every statement $\ell_i^j$: $v_m \leftarrow v_j$ and every statement
$\ell_i^j$: $v_m \leftarrow C$ are replaced by $\ell_i^j$: BEGIN $v_m \leftarrow \Omega$;
GO TO $\ell_i^{m+1}$ END.

Every statement $\ell_i^j$: IF $V_t = C$ THEN $S_1$ ELSE $S_2$ is
replaced by $\ell_i^j$: $S_1$ if $t = j$ and $\ell_i^j$: $S_2$
otherwise.

3) In copy $S^{k+1}$

Every $\text{PUSHLEFT}(Q,v)$ is replaced by $\text{PUSH}(PD_1,v)$.

Every $\text{PUSHRIGHT}(Q,v)$ is replaced by $\text{PUSH}(PD_2,v)$.

Every instruction $\ell_i^{k+1}$: $\text{POPLEFT}(Q,v_m)$ is replaced
by $\ell_i^{k+1}$: BEGIN IF $\text{EMPTYPDS}(PD_1)$ THEN
BEGIN $v_m \leftarrow \Omega$;
UNTIL $\text{EMPTYPDS}(PD_2)$ DO
BEGIN $\text{POP}(PD_2,Z)$;
$\text{PUSH}(PD_1,Z)$
END;
GO TO $\ell_i^m$
END;
ELSE $\text{POP}(PD_1,v_m)$
END.

Every statement $\ell_i^{k+1}$: $\text{POPRIGHT}(Q,v_m)$ is replaced by
$\ell_i^{k+1}$: BEGIN IF $\text{EMPTYPDS}(PD_2)$ THEN
BEGIN $v_m \leftarrow \Omega$; GO TO $\ell_i^{m+1}$ END;
ELSE $\text{POP}(PD_2,v_m)$
END.

Every statement $\ell_i^{k+1}$: IF $v_m = C$ THEN $S_1$ ELSE $S_2$ is
replaced by $\ell_i^{k+1}$: $S_2$. 
Every statement \( v_m \leftarrow C \) is transformed into \( v_m \leftarrow \Omega \).

The above translations demonstrate how to keep up with the chip \( C \) and to simulate the behavior of the queue. This establishes the inclusion \( P_{2b,0} \geq P_{1Q}(1) \) and the theorem.

Q.E.D.

(2.8) **Corollary:** \( P_{2b,0} \geq P_{1Q}(1) \).

This result does not say anything about the power of types of queues simpler than the D-structure, but it can be proved that the most restricted one-chip queue formulation is at least as powerful as the least-restricted zero-chip design. Stated in information flow terms, this is the same as saying that a single data structure with only FIFO data flow and with one chip (place marker) is at least as powerful as a single data structure with either FIFO or LIFO (or mixed) flow and no chips. The following lemma outlines the proof of this conclusion.

(2.9) **Lemma:** \( P_{1Q}(1) \geq P_{1Q}(0) \).

**Proof** By Lemma 2.2 a proof that \( P_{1Q_C}(1) \geq P_{1Q_D}(0) \) will be sufficient. If \( S \) is a schema in \( P_{1Q_D}(0) \), \( S' \) in \( P_{1Q_C}(1) \) is constructed by copying \( S \) with changes only to the queue accessing statements:

1) PUSHLEFT and POPRIGHT statements are unchanged.
2) POплеfT is handled by inserting the chip into the simulating queue and using a buffered circular shift
to obtain the element on the left-hand end of the queue.

3) **PUSHRIGHT** is simulated by inserting the chip followed by the new element and shifting circularly until the chip is shifted off the right end of the queue.

Q.E.D.

It is again fairly clear that the simple augmentation of a single one-chip queue with an ordinary pushdown store yields universality by the fact that $P_{1Q(1)}$ can handle integer arithmetic by Gödelizing the content of a fixed number $k$ of integer variables. Any value is used as a filler, and for distinct primes $p_i$, the relation $|Q| = p_1^{v_1} p_2^{v_2} \cdots p_k^{v_k}$ is maintained. Incrementation and decrementation of integer variable $v_j$ by 1 corresponds to multiplying and dividing, respectively, $|Q|$ by $p_j$; such operations can be easily carried out using the chip as a place marker and circular shifts of the queue. Theorem 10.10 of [Con72] ($P_{1(0)+N}$ is universal) then implies the following

(2.10) **Lemma:** $P_{1Q(1)+(1,0)}$ is universal.

It should be noted that the lemma is not restricted to queues of the D-type.

As the below theorem shows, the class $P_{1Q(2)}$ is universal, and it is hypothesized that this class is strictly larger than the one-chip classes of queue schemata. The basic data flow
is unchanged from other queue classes; the power increment seems to come from the added control power gained by allowing another place marker to be inserted in the data at an arbitrary point. The proof of the universality of the two chip queue is accomplished by a direct simulation which is quite long and involved. Details shown here by way of example will not be repeated in similar later proofs.

(2.11) Theorem: \( P_{\text{1Q}}(2) \) is universal.

Proof: Section 8 of [Con72] and Section 4 of [Bro72] together show that \( P_{(3b,0)} \) and \( P_{\text{AM}} \) are effectively equivalent universal classes. If \( S \) is a schema in \( P_{\text{1QD}}(2) \), a schema \( S' \) in \( P_{\text{AM}} \) can be constructed to simulate \( S \) by allocating an array to contain the contents of the queue, designating markers in \( S' \) to correspond to the two chips of \( S \), and bracketing the simulated queue with boundary flags. This argument can be straightforwardly formalized to show \( P_{\text{AM}} \geq P_{\text{1Q}}(2) \) or \( P_{(3b,0)} \geq P_{\text{1Q}}(2) \).

If, on the other hand, \( S \) is a schema in \( P_{(3b,0)} \), \( S' \) in \( P_{\text{1Q}}(2) \) can be constructed to simulate \( S \) using the queue \( Q \) and chips \( C \) and \( C' \). The body of \( S \) is used as a base, and all pds manipulations are transformed into queue operations on an encoded representation of the contents of the three pds's. The constructions use only \textsc{pushleft} and \textsc{popright}, so \( S' \) is in the class \( P_{\text{1QC}}(2) \), and the theorem is then established by Lemma 2.1.
The details of the construction involve storage of the pds contents (for pds's PD1, PD2, and PD3) by interleaved Gödelization, as explained in the introduction:

\[ \cdots \quad \Omega \quad c_1 \quad b_1 \quad a_1 \]

The first statement in \( S' \) is \( \text{PUSHLEFT}(Q,\Omega) \) to initialize the queue. The following description shows how tests for emptiness, pushes, and pops are simulated, which is sufficient to establish the result.

To test pds \( PD_j \) for emptiness the quantity 
\[ t = |Q| \mod p_j \]

is computed; \( PD_j \) is empty iff \( t \neq 0 \). Code to replace the statement

\[ \text{IF EMPTYPDS}(PD_j) \text{ THEN } S_1 \text{ ELSE } S_2 \]

is then

\begin{verbatim}
BEGIN PUSHLEFT(Q,C); POPRIGHT(Q,W); PUSHLEFT(Q,W);
POPRIGHT(Q,W);
LOOP: IF W = C THEN GO TO EMPTYJ;
PUSHLEFT(Q,W); POPRIGHT(Q,W);
IF W = C THEN GO TO EMPTYJ;
PUSHLEFT(Q,W); POPRIGHT(Q,W); \} \text{repeated} \quad p_j - 2 \text{ times}
::
IF W = C THEN GO TO NOTEMPTYJ;
PUSHLEFT(Q,W); POPRIGHT(Q,W);
GO TO LOOP;
EMPTYJ: S_1; GO TO OUT;
NOTEMPTYJ: S_2
OUT: \end{verbatim}

\( W \) is a previously unused variable; the names LOOP, EMPTYJ, NOTEMPTYJ, and OUT are clearly different for each instance of such a test statement.
To execute \( \text{POP(PD} j, v) \) the following steps are required:

a) If \( \text{Pd} j \) is empty the pop is skipped.

b) Otherwise, the elements of the simulated pds are percolated:

\[ \begin{align*}
&\text{the } j^{\text{th}} \text{ element of } Q \text{ is copied into } v; \\
&\text{the } (j+3)^{\text{rd}} \text{ element is moved to the } j^{\text{th}} \text{ spot;} \\
&\text{the } (j+6)^{\text{th}} \text{ element is moved to the } (j+3)^{\text{rd}} \text{ spot, etc.}
\end{align*} \]

c) The length of the queue is then divided by \( p_j \), with the discarded elements being taken from the padding.

Code for parts b and c (part a is a special case of the emptiness test) is as follows:

\[
\begin{align*}
PUSHLEFT(Q, C); \\
POPRI\text{GHT}(Q, W); &\quad \text{PUSHLEFT}(Q, W): \leftarrow \text{repeated } j - 1 \text{ times} \\
&\vdots \\
&\text{POPRI\text{GHT}}(Q, v); \quad \text{Desired element is obtained.} \\
&\text{LOOP1: POPRI\text{GHT}}(Q, V1); \quad \text{Elements are now percolated.} \\
&\quad \text{IF } V1 = C \text{ THEN GO TO DIVIDELENGTH; } \\
&\quad \text{POPRI\text{GHT}}(Q, V2); \quad \text{IF } V2 = C \text{ THEN GO TO RESTORE1;} \\
&\quad \text{POPRI\text{GHT}}(Q, V3); \quad \text{IF } V3 = C \text{ THEN GO TO RESTORE2;} \\
&\quad \text{PUSHLEFT}(Q, V3); \quad \text{PUSHLEFT}(Q, V1); \quad \text{PUSHLEFT}(Q, V2); \\
&\quad \text{GO TO LOOP1;}
\end{align*}
\]

\[
\begin{align*}
&\text{RESTORE1: PUSHLEFT}(Q, V1); \quad \text{Elements in buffer are} \\
&\quad \text{GO TO DIVIDELENGTH; restored; this replacement} \\
&\quad \text{becomes necessary if } p_i \neq 3 \\
&\text{RESTORE2: PUSHLEFT}(Q, V1); \quad \text{for any } i \text{ corresponding} \\
&\quad \text{PUSHLEFT}(Q, V2); \quad \text{to a nonempty pds.}
\end{align*}
\]

\[
\text{DIVIDELENGTH: This division is accomplished by using the two chips, which, logically, start from} \\
\text{opposite ends of the queue and are moved toward each other. The chip } C \text{ beginning} \\
\text{at the padded end of the queue moves } p_j - 1 \\
\text{steps at a time (and the elements it} \\
\text{crosses are discarded), while its counterpart} \\
\text{chip } C' \text{ moves only one step at a time. When} \\
\text{the chips meet, the queue length has been} \\
\text{divided by } p_j.
\]
PUSHLEFT(Q,C); PUSHLEFT(Q,C'); POPRIGHT(Q,V1);
UNTIL V1 = C DO
BEGIN POPRIGHT(Q,V2); ...; POPRIGHT(Q,VP_j);
A buffer is set up.
UNTIL VP_j = C DO
BEGIN PUSHLEFT(Q,V1);
V1 + V2; V2 + V3; ...
V(P_j-1) + VP_j;
POPRIGHT(Q,VP_j)
END;
The buffered shift above now permits p_j - 1 elements to
be trimmed from the padding
by simply ignoring the buffer.

V + VP_j;
UNTIL V = C' DO
BEGIN PUSHLEFT(Q,V);
POPRIGHT(Q,V)
END;
POPRIGHT(Q,V1);
PUSHLEFT(Q,V1);
PUSHLEFT(Q,V);

END
POPRIGHT(Q,V);
UNTIL V = C' DO
BEGIN PUSHLEFT(Q,V);
POPRIGHT(Q,V)
END

Accomplishment of the statement PUSH(PDJ_j,v) requires a
similar sequence of operations to that for the pop:

a) Q is multiplied by p_j.
b) The new element is inserted into the jth position,
and the jth element is percolated to the (j+3)rd
spot, the (j+3)rd element to the (j+6)th spot, etc.

The multiplication is carried out using a moving marker scheme
like the one used for division in popping so that the code
for the push is the statement

\begin{verbatim}
BEGIN PUSHLEFT(Q,C); PUSHLEFT(Q,C');

POPRIGHT(Q,W); POPRIGHT(Q,V1);
UNTIL V1 = C DO
  BEGIN UNTIL V1 = C DO
    BEGIN PUSHLEFT(Q,W); W + V1;
         POPRIGHT(Q,V1)
    END;
    PUSHLEFT(Q,V1); PUSHLEFT(Q,W);
    POPRIGHT(Q,V1);
    UNTIL V1 = C' DO
      BEGIN PUSHLEFT(Q,V1); POPRIGHT(Q,V1)
      END;
      PUSHLEFT(Q,V1);
      POPRIGHT(Q,V1)
  END;

The above section multiplies the length of the queue by $p$, using a moving marker scheme as in the division, so that the added elements are inserted at the logical end of the data.

POPRIGHT(Q,V1); PUSHLEFT(Q,V1); \leftarrow repeated $j - 1$ times

\vdots

POPRIGHT(Q,W); PUSHLEFT(Q,V);

LOOP: POPRIGHT(Q,V1);
IF V1 = C' THEN GO TO OUT;
PUSHLEFT(Q,V1); POPRIGHT(Q,V1);
IF V1 = C' THEN GO TO OUT;
PUSHLEFT(Q,V1); POPRIGHT(Q,V1);
IF V1 = C' THEN GO TO OUT;
PUSHLEFT(Q,W);
W + V1;
GO TO LOOP

OUT:
END
\end{verbatim}

Q.E.D.

The fact that $P_{1D}(2) \equiv P_{(3b,0)}$ follows at once from the result that $P_{1D}(1) \equiv P_{(2b,0)}$ (Theorem 2.7) and from Theorem 4.11 of [Bro72], which shows that $P_{(2b,0)}$ with one chip is universal. The theorem just proved is somewhat stronger,
however, in that the queue of the class does not have to be a deque.

2.4 **Multiple Queues with Chips**

The single queue with its inherent FIFO data flow has thus been shown to be a powerful structure, which essentially lacks only two place markers to be universal. A two-queue class offers even more flexibility of information movement because of the potential interaction between the two queues. One might suspect that adding only one chip to a two-queue class would yield universality since the two queues can be used in tandem so that the end of the first queue serves as (logically) one of the markers in a simulation patterned after that used in Theorem 2.11. That is

\[
\begin{align*}
Q1: & \quad \rightarrow \begin{array}{c}
data_1 \\
data_2 \\
\cdots \\
data_i \\
\end{array} \\
Q2: & \quad \rightarrow \begin{array}{c}
data_{i+1} \\
data_{i+2} \\
\cdots \\
data_n \\
\end{array}
\end{align*}
\]

In the direct simulation used to prove the following theorem just such an approach is taken.

**(2.12) Theorem:** \(\mathcal{P}_{2Q}(1)\) is universal.

**Proof** The class \(\mathcal{P}_{2Q}(1)\) will be shown equivalent to the universal class \(\mathcal{P}_{(3b,0)}\). That \(\mathcal{P}_{2Q}(1) \subseteq \mathcal{P}_{(3b,0)}\) follows at once from the equivalence of \(\mathcal{P}_{(3b,0)}\) and \(\mathcal{P}_{AM}\) (Section 8 of [Con72] and Section 4 of [Bro72]), and the obvious simulation of a schema in \(\mathcal{P}_{2Q}(1)\) using the arrays
and markers of $P_{AM}$.

The inclusion in the other direction is demonstrated by a direct simulation which operates using an interleaved Gödel-ization of the contents of the pds's of $S$, a schema in $P_{(3b,0)}$. The simulating schema $S'$ is constructed directly from $S$ by modifying emptiness tests, pushes, and pops as described below. The statement $PUSHLEFT(Q1,\Omega)$ is inserted at the beginning of $S'$ to initialize the simulated storage.

1) The canonical configuration of the queues in $S'$ is for $Q1$ to contain all the pds elements, so pds emptiness tests can be accomplished using exactly the same approach and code as in Theorem 2.11.

2) In place of each statement $PUSH(PDj,v)$ instructions are inserted to multiply the length of $Q1$ by $p_j$ and then percolate elements downward as in Theorem 2.11. The length multiplication is the only particular different from the earlier proof, and it is accomplished by accumulating padding in $Q2$ by inserting $p_j - 1$ elements there for every element of $Q1$ (which is examined via a circular shift with the place marker). $Q1$ and $Q2$ are then "glued together" and the shifting of elements follows. The code below illustrates the multiplication method.
PUSHLEFT(Q1,C); POPRIGHT(Q1,W);
UNTIL W = C DO
    BEGIN PUSHLEFT(Q1,W);
        PUSHLEFT(Q2,OMEGA); \(\rightarrow\) repeated \(p_j - 1\) times
    \(
\vdots
    \)
        POPRIGHT(Q1,W)
    END;

PUSHLEFT(Q2,C); POPRIGHT(Q2,W);
UNTIL W = C DO
    BEGIN PUSHLEFT(Q1,W); POPRIGHT(Q2,W) END

3) For each POP(PDJ,v) in S is substituted a compound statement which first tests PDj for emptiness in the standard way and does nothing if the pds is empty. Otherwise, the elements of PDj are percolated upward and \(|Q1|\) is divided by \(p_j\). The division, which is accomplished by copying the length of Q1 into Q2 (using C), computing \(|Q2|/p_j\) (expressed as the length of Q2), and then taking the first \(|Q2|\) elements of Q1, is shown below; the percolation is analogous to the process shown in the proof of Theorem 2.11.

\[
\begin{align*}
&PUSHLEFT(Q1,C); POPRIGHT(Q1,W); \\
&\text{UNTIL } W = C \text{ DO} \\
&\quad \begin{align*}
&\text{BEGIN PUSHLEFT(Q2,OMEGA);} \\
&\quad \text{PUSHLEFT(Q1,W);} \\
&\quad \text{POPRIGHT(Q1,W)}
&\end{align*} \\
&\quad |Q1| \text{ is duplicated in Q2.}
\end{align*}
\]

\[
\begin{align*}
&PUSHLEFT(Q2,C); POPRIGHT(Q2,W); \\
&\text{UNTIL } W = C \text{ DO} \\
&\quad \begin{align*}
&\text{BEGIN PUSHLEFT(Q2,OMEGA);} \\
&\quad \text{POPRIGHT(Q2,W); }\leftarrow \text{ repeated } \quad p_j \text{ times}
&\end{align*}
\end{align*}
\]
\[
\text{PUSHLEFT}(Q2,C); \; \text{POPRIGHT}(Q2,W); \\
\text{UNTIL } W = C \text{ DO} \\
\text{BEGIN} \; \text{POPRIGHT}(Q1,W); \; \text{PUSHLEFT}(Q2,W); \; \text{POPRIGHT}(Q2,W) \\
\text{END}
\]

The first \(|Q2|\) elements of \(Q1\) are selected out.

\[
\text{PUSHLEFT}(Q1,C); \; \text{POPRIGHT}(Q1,W); \\
\text{UNTIL } W = C \text{ DO} \; \text{POPRIGHT}(Q1,W); \\
\text{END}
\]

The remainder of \(Q1\) is discarded.

\[
\text{PUSHLEFT}(Q2,C); \; \text{POPRIGHT}(Q2,W); \\
\text{UNTIL } W = C \text{ DO} \\
\text{BEGIN} \; \text{PUSHLEFT}(Q1,W); \; \text{POPRIGHT}(Q2,W) \\
\text{END}
\]

The elements are restored to \(Q1\).

Since only \text{PUSHLEFT} and \text{POPRIGHT} have been used in the substitutions, the simulations are valid for all queue designs.

\(\Box\)
(2.13) **Theorem:** \( P_{1QZ}(0)b \equiv P_{1QZ}(0) \) for \( Z = A, B, C, D \).

**Proof** The proof involves outlining the construction of a locator in \( P_{1Q}(0) \) for a schema \( S \) in \( P_{1QZ}(0)b \). If the locator finds both true and false arguments for a predicate, the pseudo markers that result can be used in a straightforward way to test the queue for emptiness; otherwise, as shown below, the internal state of the schema has sufficient capability to remember whether the queue is empty or not.

If the autonomous behavior of \( S \), which has \( |S| \) statements and \( t \) push statements, is considered, either \( S \) will halt with no predicate change after a finite number of statements have been executed, or it will loop, either forever or until some predicate changes value. To see this, one must exclude the effect of the bottom test in causing halting behavior after unboundedly many statements (in autonomous execution).

Under autonomous behavior the only things that can change the course of a computation from linearity are simple GO TO statements and the change of the emptiness predicate. The maximum queue size before some statement is repeated is \( t \), where \( t \leq |S| \) is the total number of push statements in \( S \). The first time that the queue size exceeds \( t \), therefore, some statement \( s \) has been executed twice since the last time the queue was empty. Furthermore, \( |Q| \) at the first execution of \( s \) was less than \( |Q| \) at the second execution. But this implies that \( S \) is in a loop such that \( |Q| \) is
increasing over the loop, and therefore that the emptiness test cannot terminate the loop. Between executions of \( |Q| \) cannot shrink to zero because that would imply that a predicate changes to "turn the growth around" from positive to negative.

Therefore in autonomous behavior the bottom test can change value only as long as \( |Q| \) has never exceeded \( t \) (afterward, it is uniformly false unless a predicate changes); replications of the schema can thus be employed to keep up with the bottom test in the range of queue sizes where it can change value. Once a predicate changes, the derived marker can be used to simulate the bottom test of \( S \).

Q.E.D.

The proof that bottom tests do not increase the power of one-chip classes takes a slightly different approach since the chip is available to be used in the simulation. For a schema \( S \) in \( P_{1Q_Z}(1) \) a simulating schema \( S' \) by replication of copies of \( S \) keeps track of whether the chip \( C \) is in the queue or not. If it is, the queue cannot be empty; otherwise the emptiness test can be performed in a straightforward way using the chip.

(2.14) **Theorem:** \( P_{1Q_Z}(1) \) \( \equiv \) \( P_{1Q_Z}(1) \) for \( Z = A, B, C, D \).

**Proof** If \( S \) has \( k \) simple variables \( v_1, \ldots, v_k \), \( S' \) is constructed from \( k + 2 \) copies of \( S \), denoted \( S^0, S^1, \ldots, S^{k+1} \), in which each statement is labeled using labels \( \lambda^j_i \), where \( j \) is the copy index and \( i \) is the index
of the statement in $S$. Execution is in copy $S^m$ if the chip resides in variable $v^m$, in copy $S^0$ if the chip has not been used, and in copy $S^{k+1}$ if the chip is in the queue. The following changes to the copies link them together:

1) In copy $S^0$

Each statement $\ell^0_i: v_m + C$ is changed to

$$\ell^0_i: \text{BEGIN } v_m + C; \text{ GO TO } \ell^m_{i+1} \text{ END.}$$

Each statement $\ell^0_i: \text{PUSH\_\_\_\_}(Q,C)$ is changed to

$$\ell^0_i: \text{BEGIN } \text{PUSH\_\_\_\_}(Q,C); \text{ GO TO } \ell^{k+1}_{i+1} \text{ END.}$$

Each statement $\ell^0_i: \text{IF EMPTY\_\_\_\_\_\_}(Q) \text{ THEN } S_1 \text{ ELSE } S_2$

is changed to

$$\ell^0_i: \text{BEGIN } \text{PUSH\_\_\_\_\_\LEFT}(Q,C); \text{ POP\_\_\_\_\_\RIGHT}(Q,W); \text{ IF } W = C \text{ THEN } S_1 \text{ ELSE BEGIN } \text{UNTIL } W = C \text{ DO} \text{ BEGIN } \text{PUSH\_\_\_\_\_\LEFT}(Q,W); \text{ POP\_\_\_\_\_\RIGHT}(Q,W) \text{ END; } S_2 \text{ END; } \text{ END.}$$

where $W$ is a new simple variable.

2) In copy $S^j$, for $j = 1, \ldots, k$

Each statement $\ell^j_i: v_m + C$ is changed to

$$\ell^j_i: \text{BEGIN } v_m + C; \text{ GO TO } \ell^m_{i+1} \text{ END.}$$

Each statement $\ell^j_i: v_m + v_j$ is changed to

$$\ell^j_i: \text{BEGIN } v_m + v_j; \text{ GO TO } \ell^m_{i+1} \text{ END.}$$

Each statement $\ell^j_i: \text{PUSH\_\_\_\_}(Q,C)$ is changed to

$$\ell^j_i: \text{BEGIN } \text{PUSH\_\_\_\_}(Q,C); \text{ GO TO } \ell^{k+1}_{i+1} \text{ END.}$$
Each statement $\ell^j_i$: \textsc{Push}(Q,v_j) is changed to

$\ell^j_i$: BEGIN \textsc{Push}(Q,v_j); GO TO $\ell^{k+1}_{i+1}$ END.

Each statement $\ell^j_i$: IF \textsc{EmptyQueue}(Q) THEN S1 ELSE S2 is changed to

$\ell^j_i$: BEGIN \textsc{PushLeft}(Q,C); \textsc{PopRight}(Q,W);
IF W = C THEN BEGIN \textsc{v}_j \leftarrow C; \textsc{S1} END
ELSE
BEGIN
UNTIL W = C DO
BEGIN \textsc{PushLeft}(Q,W);
\textsc{PopRight}(Q,W)
END;
\textsc{v}_j \leftarrow C; \textsc{S2}
END;

$\ell^j_i$: END.

Each statement $\ell^j_i$: \textsc{Pop}(Q,v_j) is changed to

$\ell^j_i$: BEGIN \textsc{Pop}(Q,v_j); GO TO $\ell^0_{i+1}$ END.

3) In copy $s^{k+1}$

Each statement $\ell^{k+1}_i$: $v_j \leftarrow C$ and each statement

$\ell^{k+1}_i$: \textsc{Push}(Q,C) is changed to a labeled null statement.

Each statement $\ell^{k+1}_i$: \textsc{Pop}(Q,v_j) is changed to

$\ell^{k+1}_i$: BEGIN \textsc{Pop}(Q,v_j);
IF $v_j = C$ THEN GO TO $\ell^j_i$
END.

Each statement $\ell^{k+1}_i$: IF \textsc{EmptyQueue}(Q) THEN S1 ELSE S2 is changed to $\ell^{k+1}_i$: S2.

The above constructions simulate \textsc{EmptyQueue} tests and therefore demonstrate the desired inclusion.

\textsc{Q.E.D.}
of the statement in S. Execution is in copy \( S^m \) if the chip resides in variable \( v_m \), in copy \( S^0 \) if the chip has not been used, and in copy \( S^{k+1} \) if the chip is in the queue. The following changes to the copies link them together:

1) In copy \( S^0 \)

Each statement \( \ell_i^0: v_m \rightarrow C \) is changed to

\[
\ell_i^0: \text{BEGIN } v_m \rightarrow C; \text{ GO TO } \ell_{i+1}^m \text{ END.}
\]

Each statement \( \ell_i^0: \text{PUSH}(Q,C) \) is changed to

\[
\ell_i^0: \text{BEGIN PUSH}(Q,C); \text{ GO TO } \ell_{i+1}^{k+1} \text{ END.}
\]

Each statement \( \ell_i^0: \text{IF EMPTYQUEUE}(Q) \text{ THEN S1 ELSE S2} \)

is changed to

\[
\ell_i^0: \text{BEGIN PUSHLEFT}(Q,C); \text{ POPRIGHT}(Q,W); \text{ IF } W = C \text{ THEN S1}
\]
\[
\text{ELSE BEGIN UNTIL } W = C \text{ DO}
\]
\[
\text{BEGIN PUSHLEFT}(Q,W); \text{ POPRIGHT}(Q,W)
\]
\[
\text{END;}
\]
\[
\text{S2}
\]
\[
\text{END;}
\]

where \( W \) is a new simple variable.

2) In copy \( S^j \), for \( j = 1, \ldots, k \)

Each statement \( \ell_i^j: v_m \rightarrow C \) is changed to

\[
\ell_i^j: \text{BEGIN } v_m \rightarrow C; \text{ GO TO } \ell_{i+1}^m \text{ END.}
\]

Each statement \( \ell_i^j: v_m \rightarrow v_j \) is changed to

\[
\ell_i^j: \text{BEGIN } v_m \rightarrow v_j; \text{ GO TO } \ell_{i+1}^j \text{ END.}
\]

Each statement \( \ell_i^j: \text{PUSH}(Q,C) \) is changed to

\[
\ell_i^j: \text{BEGIN PUSH}(Q,C); \text{ GO TO } \ell_{i+1}^{k+1} \text{ END.}
\]
Each statement \( \ell^j_i \): PUSH\(\)(Q,v\(\_\)j) is changed to
\[
\ell^j_i: \text{BEGIN} \quad \text{PUSH\(\)}(Q,v\_j)\text{;} \quad \text{GO TO} \quad \ell^{k+1}_{i+1} \quad \text{END.}
\]

Each statement \( \ell^j_i \): IF EMPTYQUEUE(Q) THEN S\(\)1 ELSE S\(\)2 is changed to
\[
\ell^j_i: \text{BEGIN} \quad \text{PUSHLEFT}(Q,C)\text{;} \quad \text{POPRIGHT}(Q,W)\text{;} \quad \text{IF} \quad W = C \quad \text{THEN BEGIN} \quad v\_j \leftarrow C; \quad \text{S}1 \quad \text{END ELSE}
\]
\[
\quad \text{BEGIN \quad UNTIL} \quad W = C \quad \text{DO}
\quad \begin{align*}
\quad & \text{BEGIN} \quad \text{PUSHLEFT}(Q,W)\text{;} \\
\quad & \quad \text{POPRIGHT}(Q,W) \\
\quad & \quad \text{END;}
\quad & v\_j \leftarrow C; \quad \text{S}2 \\
\quad & \text{END;}
\end{align*}
\]
\[
\text{END.}
\]

Each statement \( \ell^j_i \): POP\(\)(Q,v\(\_\)j) is changed to
\[
\ell^j_i: \text{BEGIN} \quad \text{POPRIGHT}(Q,v\_j)\text{;} \quad \text{GO TO} \quad \ell^0_{i+1} \quad \text{END.}
\]

3) In copy \( s^{k+1} \)
Each statement \( \ell^{k+1}_i \): \( v\_j \leftarrow C \) and each statement
\( \ell^{k+1}_i \): PUSH\(\)(Q,C) is changed to a labeled null statement.

Each statement \( \ell^{k+1}_i \): POP\(\)(Q,v\(\_\)j) is changed to
\[
\ell^{k+1}_i: \text{BEGIN} \quad \text{POPRIGHT}(Q,v\_j)\text{;} \\
\quad \text{IF} \quad v\_j = C \quad \text{THEN GO TO} \quad \ell^j_{i+1} \quad \text{END.}
\]

Each statement \( \ell^{k+1}_i \): IF EMPTYQUEUE(Q) THEN S\(\)1 ELSE S\(\)2 is changed to \( \ell^{k+1}_i \): S\(\)2.

The above constructions simulate EMPTYQUEUE tests and therefore demonstrate the desired inclusion. 

Q.E.D.
of the statement in $S$. Execution is in copy $S^m$ if the chip resides in variable $v_m$, in copy $S^0$ if the chip has not been used, and in copy $S^{k+1}$ if the chip is in the queue.

The following changes to the copies link them together:

1) In copy $S^0$

Each statement $\ell_i^0: v_m + C$ is changed to

$\ell_i^0$: BEGIN $v_m + C$; GO TO $\ell_{i+1}^m$ END.

Each statement $\ell_i^0$: PUSH__(Q,C) is changed to

$\ell_i^0$: BEGIN PUSH__(Q,C); GO TO $\ell_{i+1}^{k+1}$ END.

Each statement $\ell_i^0$: IF EMPTYQUEUE(Q) THEN S1 ELSE S2

is changed to

$\ell_i^0$: BEGIN PUSHLEFT(Q,C); POPRIGHT(Q,W);

IF W = C THEN S1

ELSE BEGIN UNTIL W = C DO

BEGIN PUSHLEFT(Q,W);

END;

POPRIGHT(Q,W)

S2

END;

END;

where $W$ is a new simple variable.

2) In copy $S^j$, for $j = 1, \ldots, k$

Each statement $\ell_i^j: v_m + C$ is changed to

$\ell_i^j$: BEGIN $v_m + C$; GO TO $\ell_{i+1}^m$ END.

Each statement $\ell_i^j: v_m + v_j$ is changed to

$\ell_i^j$: BEGIN $v_m + v_j$; GO TO $\ell_{i+1}^m$ END.

Each statement $\ell_i^j$: PUSH__(Q,C) is changed to

$\ell_i^j$: BEGIN PUSH__(Q,C); GO TO $\ell_{i+1}^{k+1}$ END.
Each statement $\ell_i^j : \text{PUSH}__(Q,v_j)$ is changed to

$\ell_i^j : \text{BEGIN} \quad \text{PUSH}__(Q,v_j) ; \text{GO TO} \quad \ell_{i+1}^{k+1} \quad \text{END}.$

Each statement $\ell_i^j : \text{IF EMPTYQUEUE}(Q) \text{ THEN } S_1 \text{ ELSE } S_2$

is changed to

$\ell_i^j : \text{BEGIN} \quad \text{PUSHLEFT}(Q,C) ; \text{POPRIGHT}(Q,W) ;$

$\quad \text{IF } W = C \text{ THEN BEGIN } v_j + C ; S_1 \text{ END ELSE}$

$\quad \text{BEGIN}$

$\quad \text{UNTIL } W = C \text{ DO}$

$\quad \text{BEGIN} \quad \text{PUSHLEFT}(Q,W) ;$

$\quad \text{POPRIGHT}(Q,W)$

$\quad \text{END} ;$

$\quad v_j + C ; S_2$

$\text{END} ;$

$\text{END}.$

Each statement $\ell_i^j : \text{POP}__(Q,v_j)$ is changed to

$\ell_i^j : \text{BEGIN} \quad \text{POP}__(Q,v_j) ; \text{GO TO} \quad \ell_{i+1}^0 \quad \text{END}.$

3) In copy $s^{k+1}$

Each statement $\ell_i^{k+1} : v_j + C$ and each statement

$\ell_i^{k+1} : \text{PUSH}__(Q,C)$ is changed to a labeled null statement.

Each statement $\ell_i^{k+1} : \text{POP}__(Q,v_j)$ is changed to

$\ell_i^{k+1} : \text{BEGIN} \quad \text{POP}__(Q,v_j) ;$

$\quad \text{IF } v_j = C \text{ THEN GO TO } \ell_{i+1}^j \quad \text{END}.$

Each statement $\ell_i^{k+1} : \text{IF EMPTYQUEUE}(Q) \text{ THEN } S_1 \text{ ELSE } S_2$

is changed to $\ell_i^{k+1} : S_2.$

The above constructions simulate $\text{EMPTYQUEUE}$ tests and therefore demonstrate the desired inclusion.  

Q.E.D.
The situation with respect to bottom tests becomes more opaque in the case of schemata with two queues, no chips, and bottom tests. As long as at least one of the queues is not of type C, the configuration is universal, as the following direct proof shows. The class \( P_{2Q_C(0)b} \), however, is conjectured to be subuniversal. Intuitively, the reason for this is the inability of the control and data structures to reverse an arbitrary string, and that capability is needed to perform the conventional simulations which prove universality.

(2.15) **Theorem:** \( P_{1Q_C(0)b + 1Q_B(0)b} \) is universal.

**Proof** Universality follows at once from showing how a schema \( S \) in \( P_{1Q_C(0)b + 1Q_B(0)b} \) can use interleaved Gödelization to simulate three bottom-testable pds's. As in earlier proofs, it is sufficient to demonstrate how to perform emptiness tests, pushes, and pops.

**Emptiness tests:** The usual method (cf. Theorem 2.11) of computing the queue length modulo \( p_j \) can be applied; QC (the canonical location where the elements are kept) is simply shifted to QB as the modulus is taken, and then QC is restored from QB.

**Pushes:** A rather interesting ravel is employed to multiply the length of the queue and to cause padding to appear at the end of the elements. QC is shifted to QB, with a padding element being inserted after
every data element; \( Q_B \) is then restored to \( QC \) by taking every other element of \( QB \) (beginning with the first) for insertion into \( QC \), and recirculating each companion element into \( QB \). The padding is reordered, but winds up at the end of the data elements. Code for this process is the following:

\[
\text{UNTIL EMPTYQUEUE}(QC) \text{ DO} \\
\begin{align*}
\text{BEGIN} & \quad \text{POPRIGHT}(QC,W); \\
& \quad \text{PUSHLEFT}(QB,W); \\
& \quad \text{PUSHLEFT}(QB,OMEGA) \\
\text{END;} \\
\text{POPRIGHT}(QB,W); \\
\text{UNTIL EMPTYQUEUE}(QB) \text{ DO} \\
\begin{align*}
\text{BEGIN} & \quad \text{PUSHLEFT}(QC,W); \\
& \quad \text{POPRIGHT}(QB,W); \\
& \quad \text{PUSHLEFT}(QB,W); \\
& \quad \text{POPRIGHT}(QB,W) \\
\text{END;} \\
\text{PUSHLEFT}(QC,W); \\
\end{align*}
\]

The percolation of elements and insertion of the new value is then carried out as in earlier proofs.
The example below shows how the raveling process works for a queue containing five elements a, b, c, d, e; padding is denoted by 1, 2, 3, 4, 5.

Initially

e d c b a

After padding insertion

<empty>

5 e 4 d 3 c 2 b 1 a

Steps in padding shift

<table>
<thead>
<tr>
<th></th>
<th>1 5 e 4 d 3 c 2 b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>b a</td>
</tr>
<tr>
<td>d</td>
<td>c b a</td>
</tr>
<tr>
<td>e</td>
<td>d c b a</td>
</tr>
<tr>
<td>1</td>
<td>e d c b a</td>
</tr>
<tr>
<td>l</td>
<td>e d c b a</td>
</tr>
<tr>
<td>3</td>
<td>l e d c b a</td>
</tr>
<tr>
<td>5</td>
<td>3 l e d c b a</td>
</tr>
<tr>
<td>4</td>
<td>5 3 l e d c b a</td>
</tr>
<tr>
<td>2</td>
<td>4 5 3 l e d c b a</td>
</tr>
</tbody>
</table>

Pops: An emptiness test is first performed, and if the simulated pds is nonempty, elements are percolated as in earlier proofs. The inverse of the push ravel process then allows removal of the padding using only C-type operations, provided the queue contents are first reversed. The code below illustrates the padding removal:
UNTIL EMPTYQUEUE(QC) DO
BEGIN
    POPRIGHT(QC,W);
    PUSHRIGHT(QB,W)
END;

The queue contents are reversed.

POPRIGHT(QB,W);
PUSHLEFT(QC,W);
PUSHLEFT(QC,Z);
PUSHLEFT(QC,W);
PUSHLEFT(QC,W);
PUSHRIGHT(QB,W)

UNTIL EMPTYQUEUE(QB) DO
BEGIN
    POPRIGHT(QC,Z);
    PUSHLEFT(QC,Z);
PUSHLEFT(QC,W);
PUSHLEFT(QC,W);
PUSHRIGHT(QB,W)
END;

The unraveling is performed.

PUSHRIGHT(QB,W);
UNTIL EMPTYQUEUE(QC) DO
BEGIN
    POPRIGHT(QC,W);
PUSHRIGHT(QB,W)
END;

UNTIL EMPTYQUEUE(QB) DO
BEGIN
    POPRIGHT(QB,W);
PUSHRIGHT(QB,W)
END;

The contents are reversed and restored to QC.

Q.E.D.

It should be clear that an A-type queue is also able to perform the two required reversals for the pop simulation.

The class \( P_{2QC}(0)b \) is hypothesized not to be able to reverse an arbitrary string, which capability is equivalent to being able to support LIFO data flow as well as FIFO data flow. Schemas of such a class can search the Herbrand universe for the subcase of monadic predicates and dyadic functions, but the search technique does not appear to be extendible to functions of higher rank.

(2.16) **Theorem:** \( P_{2QC}(0)b \) can search the Herbrand universe for monadic predicates and dyadic functions.
Proof  Without loss of generality, the following will consider one predicate p, one function f(·,·), and one domain value x. Then the following schema performs the required computation by using the methods of the push simulation in Theorem 2.15.

\[(x): \text{PUSHLEFT}(Q_1, f(x,x)); \text{PUSHLEFT}(Q_1, x);\]
\[\text{TOP}: \text{POPRIGHT}(Q_1, W); \text{UNTIL EMPTYQUEUE}(Q_1) \text{ DO} \begin{array}{l}
\text{BEGIN} \text{POPRIGHT}(Q_1, V); \text{PUSHLEFT}(Q_2, V); \\
\text{IF } p(f(W, V)) \text{ THEN } \text{HALT}(x) \\
\text{ELSE } \text{PUSHLEFT}(Q_2, f(W, V)) \\
\text{END}; \\
\text{POPRIGHT}(Q_2, V); \\
\text{UNTIL EMPTYQUEUE}(Q_2) \text{ DO} \begin{array}{l}
\text{BEGIN} \text{PUSHLEFT}(Q_1, V); \text{POPRIGHT}(Q_2, V); \\
\text{PUSHLEFT}(Q_2, V); \text{POPRIGHT}(Q_2, V) \\
\text{END}; \\
\text{PUSHLEFT}(Q_1, V); \\
\text{PUSHLEFT}(Q_1, f(W, W)); \\
\text{PUSHLEFT}(Q_1, W); \\
\text{GO TO TOP};
\end{array}\]
\[\text{Q.E.D.}\]

This example, then, supports the conjecture that power to search the Herbrand universe for monadic predicates and dyadic functions is not enough for universality (though one might intuitively expect the opposite).

Addition of the bottom test to higher order queue schemata yields universality by an argument involving showing equivalence to the class \(P_{(1,0)} + \mathbb{N}\). Thus (without formal proof),
(2.17) Theorem: $P_{NQ(0)b} = P_{(3b,0)}$ for $N \geq 3$.

Queues are significant not only because of the intrinsically interesting relationship of FIFO data flow capability to the power of other flows, but also for practical reasons. First, queues provide a gross model of certain computer elements, and second, queues suggest a data structure which is "testable" (for power and convenience) on a schema level, but which has not been widely implemented on a hardware level. Asynchronous recirculating delay line memories are modeled to a degree by queues (though such memories generally have a fixed capacity, rather than the variable size defined for queues); the end of the queue is analogous to the implicit timing mark (or clock pulse) in the delay line circuitry which marks the beginning of the elements stored, and chips serve as additional explicit time markers. Tracks of a disk or drum are logically quite close to the delay lines, and hence are also related to queues from a data flow viewpoint.

To evaluate the possible utility of the queue in actual programming applications, one should first note that the device is best suited to problems in which the data must flow cyclically. One possible application is matrix multiplication and another is sorting. In the first case the
outline below is strictly a sketch of the analysis necessary for application of queues to the problem (since such a study is complex enough to call for an entire paper), and in the second case some indicative work has been done by Tarjan [Tarj72].

2.6 Example of the Application of Queues to Matrix Multiplication

A problem of interest is to compute the matrix product $C = A \cdot B$, where $A$ is an $m \times r$ matrix and $B$ is an $r \times n$ matrix. Such a product requires computation of all possible row-column inner products between the matrices. Since each inner product can be computed by a running sum of element-wise products and since each row and column must appear several times in the calculation, the problem seems well suited to the cyclic behavior possible for queues. Conventional matrix multiplication techniques using arrays must carry out a nontrivial address calculation requiring time $t_1$, say, for each data fetch, whereas an ideal queue would have a constant response time $t_2$ to supply its next element. The intent is to demonstrate that if $t_1 - t_2 > 0$ then (for large matrices, at least) the queue method enjoys a significant time advantage. Of course such a conclusion is also highly dependent on finding a suitable multiplication algorithm to use with queues.

The efficient use of the queue structure in matrix operations depends on selecting the proper ordering of matrix
elements within the queue. It is desirable to have an ordering that can be used in either the first or second position of a multiplication and that is preserved by the multiplication. A possible method is to select them diagonally from the square representation of the matrix. For example, for a $5 \times 5$ matrix $A$, values are taken beginning with $a_{11}$ and proceeding as shown by the arrows:

Expressed algebraically, the ordering is such that for a $p \times q$ matrix $A$, the $j^{th}$ element of the queue is $a_{u_jv_j}$, where

\[
    u_j = ((j - 1) \mod p) + 1 \\
    v_0 = 0 \\
    v_j = (v_{j-1} + 1) \mod (q + 1) + \\
    \text{if} \ (v_{j-1} + 1) \mod (q + 1) = 0 \text{ then } 1 \text{ else } 0 + \\
    \text{if} \ (j - 1) \mod ((p \cdot q)/\gcd(p,q)) = 0 \text{ then } 1 \text{ else } 0 \\
    \text{ for } j = 1, \ldots, p \cdot q \text{ and } \\
    \gcd = \text{greatest common divisor.}
\]
If C-type queues Q1 and Q2 contain the $m \times r$ matrix A and the $r \times n$ matrix B, respectively and if $\text{GCD}(m,n,r) = 1$, then the following circulation for matrix multiplication works very smoothly to insert the product into Q3:

1) m·n pairwise products are formed by shifting Q1 and Q2 in synchronization, and inserting the pairwise products in order into Q3.

2) Pairwise products are formed for $m·n·(r - 1)$ more times, each time with the product being added to the right and element of Q3 and the resulting sum being shifted into the left end of Q3.

3) The product then appears in Q3 in the same order as A and B, and A and B are unchanged.

For instance, the calculation for a $3 \times 2$ matrix A and a $2 \times 5$ matrix B is as follows:

$$Q1 \rightarrow \begin{array}{cccccc} a_{32} & a_{21} & a_{12} & a_{31} & a_{22} & a_{11} \end{array}$$

$$Q2 \rightarrow \begin{array}{ccccccc} b_{25} & b_{14} & b_{23} & b_{12} & b_{21} & b_{15} & b_{24} & b_{13} & b_{22} & b_{11} \end{array}$$

Step 1: The pairwise products $a_{11}·b_{11}, a_{22}·b_{22}, a_{31}·b_{13}, \ldots$ are formed and pushed, in order, onto Q3. After each multiplication the factors from A and B are recirculated into the appropriate register. This step proceeds until $3·5 = 15$ elements have been calculated.
Step 2: The pairwise multiplication and recirculation of factors continues for \(3.5 \cdot (2 - 1) = 15\) more times, except that each product is now added to the end element of \(Q3\), and the resulting sum is recirculated into \(Q3\) in place of the end element.

\[ Q3 \rightarrow \begin{array}{cccccccccc}
C_{35} & C_{24} & C_{13} & C_{32} & C_{21} & C_{15} & C_{34} & C_{23} & C_{12} & C_{31} & C_{25} & C_{14} & C_{33} & C_{22} & C_{11} \\
\end{array} \]

This method of storage and calculation could also be used profitably in array-type multiplication algorithms if a hardware modulus function were available.

For \(n \times n\) square matrices the queue formulation is considerably messier and involves the use of an additional queue as a work area. In this case the fact that \(\text{GCD}(m,n,r) = n\) (the dimensions are not relatively prime) causes the pairings to cycle before all possible conjunctions have been generated, and some meddling must be employed to overcome this difficulty. Extra shifting must be done as shown in the following algorithm:
QC (result queue) is initialized to contain $n^2$ zeros.

DO I = 1 TO n
BEGIN n elements of QA (containing matrix A) are moved to Q (the work queue) and the elements are recirculated into QA.

L: DO J = 1 TO $n^2$
BEGIN $Q \ast QB + QC + QC$, element-by-element, and QB and Q are recirculated.
END;

Q is emptied.

Every n-element group of QB is right rotated by one element.

QC is right rotated by n elements.
END

Expansions of the subalgorithms are these:

Every n-element group of QB is right rotated by one element:

DO K = 1 TO n
BEGIN POPRIGHT(QB,X);
DO M = 1 TO n - 1
BEGIN POPRIGHT(QB,Y);
PUSHLEFT(QB,Y)
END;
PUSHLEFT(QB,X)
END

QC is right rotated by n elements:

DO K = 1 TO n
BEGIN POPRIGHT(QC,X);
PUSHLEFT(QC,X)
END

The following multiplication is given as an example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 3 & 1 \\ 2 & 5 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 3 & 3 \\ 5 & 1 & 2 \end{bmatrix}$$
Initially,

QA: $\rightarrow \begin{array}{cccccccc}
5 & 4 & 3 & 2 & 1 & 2 & 2 & 3 & 1 \\
\end{array}$

QB: $\rightarrow \begin{array}{cccccccc}
1 & 2 & 1 & 5 & 3 & 0 & 2 & 3 & 1 \\
\end{array}$

$I = 1$, at the end of the loop labeled L,

Q: $\rightarrow \begin{array}{cccccccc}
2 & 3 & 1 \\
\end{array}$

QA: $\rightarrow \begin{array}{cccccccc}
2 & 3 & 1 & 5 & 4 & 3 & 2 & 1 & 2 \\
\end{array}$

QB: $\rightarrow \begin{array}{cccccccc}
1 & 2 & 1 & 5 & 3 & 0 & 2 & 3 & 1 \\
\end{array}$

QC: $\rightarrow \begin{array}{cccccccc}
2 & 6 & 1 & 10 & 9 & 0 & 4 & 9 & 1 \\
\end{array}$

$I = 2$, at the end of the loop labeled L,

Q: $\rightarrow \begin{array}{cccccccc}
2 & 1 & 2 \\
\end{array}$

QA: $\rightarrow \begin{array}{cccccccc}
2 & 1 & 2 & 2 & 3 & 1 & 5 & 4 & 3 \\
\end{array}$

QB: $\rightarrow \begin{array}{cccccccc}
1 & 1 & 2 & 0 & 5 & 3 & 1 & 2 & 3 \\
\end{array}$

QC: $\rightarrow \begin{array}{cccccccc}
6 & 10 & 5 & 2 & 11 & 7 & 12 & 11 & 6 \\
\end{array}$

$I = 3$, at the end of the loop labeled L,

Q: $\rightarrow \begin{array}{cccccccc}
5 & 4 & 3 \\
\end{array}$

QA: $\rightarrow \begin{array}{cccccccc}
5 & 4 & 3 & 2 & 1 & 2 & 2 & 3 & 1 \\
\end{array}$

QB: $\rightarrow \begin{array}{cccccccc}
2 & 1 & 1 & 3 & 0 & 5 & 3 & 1 & 2 \\
\end{array}$

QC: $\rightarrow \begin{array}{cccccccc}
22 & 15 & 9 & 21 & 10 & 20 & 17 & 15 & 13 \\
\end{array}$
Final configuration,

Q: empty

QA, QB: same as originally

QC: → 17 15 13 22 15 9 21 10 20 →

The complexity of the above algorithm implies that it is by no means a foregone conclusion that queue structures are economical or even feasible for matrix multiplication, but the topic should be investigated more fully and an attempt made to treat the general case in which \( \text{GCD}(m,n,r) \neq 1 \). The particular cyclic treatment used in the \( n \times n \) case is certainly not the only possible method of generating results, though it has the advantage of providing an element-wise ordering that is preserved under multiplication. Storage patterns other than the modulus method indicated may also be fruitful; the straightforward idea of storing \( A \) in row major order and \( B \) in column major order leads to simpler products, but requires two separate forms of matrix storage (with high intertranslation cost) and produces the result in yet a third ordering. Nonetheless, for this and similar problems with inherently cyclic data flow, the queue may be a logically feasible data structure.
CHAPTER 3. TAPE-LIKE STORAGE DEVICES

3.1 Definition of Tapes in Schemata

Two salient characteristics of the queue devices introduced in Chapter II are their capability for FIFO data flow (or mixed FIFO and LIFO flow) and the nonpersistence of information in the data structure (reading, or popping, from a queue can be accomplished only by deleting the endmost element from the structure, with, in some cases, no opportunity to restore it to the relative position it occupied before the pop). This chapter discusses devices with nondestructive read capability, but which are limited to strictly sequential access. These latter properties can be embodied in a tape-like structure which closely models the computer tapes which have been common mass storage devices since the late 1940's and early 1950's. The study below is therefore interesting both from the information flow aspect and from the standpoint of being a model of a real-world device.

As in the case of queues, the properties of the tape devices must be specified:

a) Tape modes

1) Neutral mode (NM): This is the initial mode of the tape; the head is positioned at the first cell, which initially contains a special end of file mark (not in the domain of computation). A rewind command causes the tape head to be positioned
at the first cell and neutral mode to be entered.

2) Input mode (IM): This mode signifies that a
tape is being read and is entered only form the
neutral mode. A READTAPE(T,x) causes IM to
be entered, and in this state all WRITETAPE(T,x)
commands are ignored.

3) Output mode (OM): This mode signifies that a
tape is being written and is also entered only
from the neutral mode. All READTAPE(T,x) com-
mands are ignored in this mode.

4) Each tape unit has one of the three above modes
associated with it at all times.

b) **Tape commands**

1) READTAPE(T,x): If the head of tape T is over
the end of file mark, nothing is done. Otherwise,
variable x receives a copy of the cell where
the tape head is, and the head is advanced one
cell.

2) WRITETAPE(T,x): The content of variable x is
recorded on tape T at the current head position
and the head is advanced one cell.

3) REWIND(T): An end of file mark is written at the
current head position if the tape is in OM. The
head is repositioned to the first cell of the
tape and neutral mode is entered.
4) IF ENDFILE(T) THEN S1 ELSE S2: If the tape is in OM, S2 is executed. Otherwise, if the cell at the current head position contains an end of file mark then S1 is executed, and if not, then S2 is executed.

5) All tape commands are subject to the mode restrictions listed under the section on tape modes.

Tapes can therefore be thought of as a variety of one-way-infinite Turing machine tape, subject to the strict separation of reading and writing implied by the above restrictions. Graphically, if EOF denotes the end of the file mark, a tape might be represented as

```
[ ... ] ... EOF
```

A schema class $P_k$ augmented with $k$ tapes is denoted by $P_x \text{ kTAPE}$, and $T_1, T_2,$ etc. are used as generic tape names in the proofs and discussions which follow.

3.2 One-Tape Classes with One-Way and Two-Way Reading Capability

A logical first step for investigation of these devices is to add one tape to the class $P$, and it is surprising to find that such an augmentation adds no power! $P_{1TAPE}$ is thus a class which has unbounded storage capacity, but which, because
of restrictions placed on accessing that storage, can make no real use of it. The key to this is the necessity that once a part of the storage has been read back after recording, no element of the storage can be changed without the loss of all previously recorded values. This accessing pattern allows a simulation of the tape device since all recorded values can be regenerated at any desired time.

(3.1) **Theorem:** \( P_{1TAPE} \equiv P \).

**Proof** The result is established by a direct simulation of a \( P_{1TAPE} \) schema \( S \) by a schema \( S' \) in \( P \). \( S' \) takes advantage of the fact that whenever a tape is written, it need only save the "state" at the time of the beginning of writing (the contents of all variables plus the location counter) in order to be able to regenerate the entire tape at any later time. Therefore, reading a tape can be simulated by executing from the saved state configuration until the first write (which gives the first element on the tape) and then saving the state at that point as the starting configuration to be used when the next read is simulated; detection of a rewind instruction signifies the end of the tape.

A large amount of information about the state of the tape, which of the \( k \) variables \( v_1, v_2, \ldots, v_k \) of \( S \) are involved in an operation, etc. must consequently be retained implicitly in the "finite control" of \( S' \), and this retention is most easily carried out by replicating \( S \). A multitude of such copies \( S[a,b,c,d,e] \) is required, each of which has
the same statement labeling (numbering) for the \( h \) statements of \( S \); a statement \( j \) in one of these copies is denoted by \( S[a,b,c,d,e,j] \). Not all subscripts are used in every case (many are dummy zeros), but the full complement is displayed so that the role of each in the rather tedious simulation might be somewhat clearer. An explanation of each position follows.

a: This index is used to keep track of the mode and to some extent the position of the tape; permissible values are

0: tape is in neutral mode with nothing written on it;

1: tape is in output mode (therefore at least one item has been written);

2: tape is in neutral mode and something has been written previously;

3: tape is in input mode; something has been written and a read is currently being simulated;

4: tape is in input mode; something has previously been written, but neither a read nor an endfile test is being simulated;

5: tape is in input mode; something has been written and an endfile test is being simulated.
The term "state i" as used below refers to the value of subscript a.

b: This index, used only in state 3, is employed to keep track of which variable will be filled by a simulated READTAPE; b ranges from 0 to k.

c: This index is used to record the statement number at which the last generation of the tape began. It is necessary to keep this original start point in case a tape is partially or fully read and then rewound for another read. The range of c is 0, 1, ..., h.

d: This index, used only in state 4 (but preserved and set in other states) is used to remember the label from which simulation of tape generation should proceed when the next READTAPE or end of file test is encountered; d ranges over 0, 1, ..., h.

e: This index has meaning only in states 3 and 5 and is used to recall the return address (the statement after the end of file test or READTAPE encountered in state 4) to which control is returned when the simulation of a tape operation is completed. The range of e is 0, 1, ..., h.

The modifications to each copy and the transfers between copies are relatively straightforward, though rather tedious,
and are given below:

In copy \( S[0,0,0,0,0] \)

a) All READTAPE statements are replaced by null statements.

b) All WRITETAPE statements \( S[0,0,0,0,j] \) are replaced by

\[
\text{BEGIN } y_1 \leftarrow v_1; \ldots; y_k \leftarrow v_k; x_1 \leftarrow v_1; \ldots; x_l \leftarrow v_k; \text{ GO TO } S[1,0,j,j,0,j+1] \text{ END.}
\]

The variables \( y_i \) contain the state at the beginning of tape generation; the variables \( x_i \) contain the state necessary to simulate the next READTAPE or end of file test.

c) All REWIND statements are replaced by null statements.

d) All statements IF ENDFILE(T) THEN S1 ELSE S2 are replaced by S1.

In copy \( S[1,0,m,m,0] \)

a) All READTAPE statements are replaced by null statements.

b) All WRITETAPE statements are replaced by null statements.

c) All REWIND statements \( S[1,0,m,m,0,j] \) are replaced by \text{GO TO } S[2,0,m,m,0,j+1].

d) All statements IF ENDFILE(T) THEN S1 ELSE S2 are replaced by S2.
In copy $S[2,0,m,m,0]$

a) All statements READTAPE($T,v_t$) labeled $S[2,0,m,m,0,j]$ are replaced by GO TO $S[3,t,m,m,j,m]$.

b) All WRITETAPE statements $S[2,0,m,m,0,j]$ are replaced by

\[
\text{BEGIN } y_1 \leftarrow v_1; \ldots; y_k \leftarrow v_k; x_1 \leftarrow v_1; \ldots; x_k \leftarrow v_k; \text{GO TO } S[1,0,j,j,0,j+1] \text{ END.}
\]

c) All REWIND statements are replaced by null statements.

d) All statements IF ENDFILE($T$) THEN $S_1$ ELSE $S_2$ are replaced by $S_2$.

In copy $S[3,t,m,r,j]$

a) All references to simple variables (except those added in step c below) $v_i$ are changed to new simple variables $z_i$.

b) All REEADTAPE statements are replaced by null statements.

c) All statements WRITETAPE($T,z_r$) labeled $S[3,t,m,r,j,n]$ are replaced by

\[
\text{BEGIN } v_t \leftarrow z_r; \text{GO TO } S[4,0,m,n+1,0,j+1] \text{ END.}
\]

d) All REWIND statements labeled $S[3,t,m,r,j,n]$ are replaced by GO TO $S[4,0,m,n,0,j+1]$.

e) All statements IF ENDFILE($T$) THEN $S_1$ ELSE $S_2$ are replaced by $S_2$. 
In copy $S[4,0,p,m,0]$

a) All statements $\text{READTAPE}(T,v_t)$ labeled $S[4,0,p,m,0,j]$ are replaced by $\text{GO TO } S[3,t,p,m,j,m]$. 

b) All $\text{WRITETAPE}$ statements are replaced by null statements. 

c) All $\text{REWIND}$ statements labeled $S[4,0,p,m,0,j]$ are replaced by
   
   $$ \begin{align*}
   \text{BEGIN } \; & x_1 \leftarrow y_1; \ldots; x_k \leftarrow y_k; \\
   \text{GO TO } & S[2,0,p,p,0,j+1] \; \text{END.}
   \end{align*} $$

   
   d) All statements $\text{IF ENDFILE}(T) \text{ THEN } S1 \text{ ELSE } S2$
labeled $S[4,0,p,m,0,j]$ are replaced by
   
   $$ \begin{align*}
   \text{BEGIN } \; & w_1 \leftarrow x_1; \ldots; w_k \leftarrow x_k; \text{ GO TO } S[5,0,p,m,j,m] \\
   \text{END.}
   \end{align*} $$

In copy $S[5,0,p,m,j]$

a) All references to simple variables $v_i$ are changed to new simple variables $w_i$. 

b) All $\text{READTAPE}$ statements are replaced by null statements. 

c) All $\text{WRITETAPE}$ statements are replaced by
   
   $$ \text{GO TO } S[4,0,p,m,0,j2] \text{ where } j2 \text{ is the label of } S2 \text{ in the end of file test (in state 4) which caused this branch to state 5 (statement } j \text{ is the } \text{IF statement).} $$

d) All $\text{REWIND}$ statements are replaced by
   
   $$ \text{GO TO } S[4,0,p,m,0,j1] \text{ where } j1 \text{ is the label} $$
of $S_1$ in the end of file test in state 4.

e) All statements IF ENDFILE(T) THEN $S_1$ ELSE $S_2$
are replaced by $S_2$.

Q.E.D.

Another characteristic that computer tapes sometimes possess is the ability to read tapes backwards (though the information of an individual record is always reversed so that it appears on proper sequence in memory after the read). To augment the tape model with this feature, a zeroth position which always contains an end of file marker and which is not writable is added to the tape; the REWIND command still positions the head to the first tape position. The new input instruction READTAPEBACKWARD(T,v) operates as follows: if the cell at (current head position) — 1 contains an end of file mark, nothing is done; otherwise the head position is decremented by one cell and the content of the cell then pointed to is placed in variable v. The ENDFILE test definition is broadened so that (in input mode) if the last previous I/O operation on tape T was a READTAPE, then ENDFILE(T) is true iff the cell at the current head position contains an end of file mark. If the last I/O operation was a READTAPEBACKWARD, then ENDFILE(T) is true iff the cell at (current head position) — 1 contains an end of file mark. The addition of $k$ such two-way tapes to a class $P_x$ is denoted by $P_x^{kTWOWAYTAPE}$. 
The original tape model corresponds to a FIFO information flow with the additional properties of rereadability and prohibition of interspersed reading and writing. Two-way tapes add the possibility of mixed FIFO and LIFO access with the same two restrictions. Once again it is found that adding this type of tape to P schemata gives no power increase. A construction which is based on ideas used in Paterson and Hewitt's discussion of the fact that every linearly recursive schema can be computed by a schema in P [Pat70] is used to show this result.

As an aid to understanding the method used in the subsequent proof, the following example, illustrating the approach, is presented. F is a functional defined by

$$F(x,p,f,g) = g(x, g(f(x), g(f^2(x), \ldots, g(f^{j-1}(x), f^j(x))\ldots)))$$

where \( j \) is the smallest integer \( \geq 1 \)

such that \( p(f^j(x)) = \text{true} \ (f^0(x) \equiv x) \).

= \text{x, if } j = 0.

= undefined, if no such \( j \) exists.

This is equivalent to the recursive formulation

$$h(x) = \text{if } p(x) \text{ then } x \text{ else } g(x, h(x)),$$

At first blush it might seem that such a functional is not computable in P because of the necessity to "back up" in computing the \( g \) composition. However, the functional can also be expressed as
of S1 in the end of file test in state 4.
e) All statements IF ENDFILE(T) THEN S1 ELSE S2 are replaced by S2.

Q.E.D.

Another characteristic that computer tapes sometimes possess is the ability to read tapes backwards (though the information of an individual record is always reversed so that it appears on proper sequence in memory after the read). To augment the tape model with this feature, a zeroth position which always contains an end of file marker and which is not writable is added to the tape; the REWIND command still positions the head to the first tape position. The new input instruction READTAPEBACKWARD(T,v) operates as follows: if the cell at (current head position) — 1 contains an end of file mark, nothing is done; otherwise the head position is decremented by one cell and the content of the cell then pointed to is placed in variable v. The ENDFILE test definition is broadened so that (in input mode) if the last previous I/O operation on tape T was a READTAPE, then ENDFILE(T) is true iff the cell at the current head position contains an end of file mark. If the last I/O operation was a READTAPEBACKWARD, then ENDFILE(T) is true iff the cell at (current head position) — 1 contains an end of file mark. The addition of k such two-way tapes to a class \( P_x \) is denoted by \( P_x kTWOWAYTAPE \)
The original tape model corresponds to a FIFO information flow with the additional properties of rereadability and prohibition of interspersed reading and writing. Two-way tapes add the possibility of mixed FIFO and LIFO access with the same two restrictions. Once again it is found that adding this type of tape to \( P \) schemata gives no power increase. A construction which is based on ideas used in Paterson and Hewitt's discussion of the fact that every linearly recursive schema can be computed by a schema in \( P \) [Pat70] is used to show this result.

As an aid to understanding the method used in the subsequent proof, the following example, illustrating the approach, is presented. \( F \) is a functional defined by

\[
F(x, p, f, g) = g(x, g(f(x), g(f^2(x), \ldots, g(f^{j-1}(x), f^j(x))\ldots))),
\]

where \( j \) is the smallest integer \( \geq 1 \) such that \( p(f^j(x)) = \text{true} \) \( (f^0(x) \equiv x) \).

\[
= x, \text{ if } j = 0.
\]

\[
= \text{undefined}, \text{ if no such } j \text{ exists.}
\]

This is equivalent to the recursive formulation

\[
h(x) = \text{if } p(x) \text{ then } x \text{ else } g(x, h(x));
\]

At first blush it might seem that such a functional is not computable in \( P \) because of the necessity to "back up" in computing the \( g \) composition. However, the functional can also be expressed as
of $S_1$ in the end of file test in state 4.

e) All statements IF ENDFILE(T) THEN $S_1$ ELSE $S_2$
are replaced by $S_2$.

Q.E.D.

Another characteristic that computer tapes sometimes
possess is the ability to read tapes backwards (though the
information of an individual record is always reversed so
that it appears on proper sequence in memory after the read).
To augment the tape model with this feature, a zeroth posi-
tion which always contains an end of file marker and which
is not writable is added to the tape; the REWIND command
still positions the head to the first tape position. The new
input instruction READTAPEBACKWARD(T,v) operates as follows:
if the cell at (current head position) — 1 contains an end
of file mark, nothing is done; otherwise the head position
is decremented by one cell and the content of the cell then
pointed to is placed in variable v. The ENDFILE test
definition is broadened so that (in input mode) if the last
previous I/O operation on tape T was a READTAPE, then
ENDFILE(T) is true iff the cell at the current head position
contains an end of file mark. If the last I/O operation was
a READTAPEBACKWARD, then ENDFILE(T) is true iff the cell
at (current head position) — 1 contains an end of file mark.
The addition of k such two-way tapes to a class $P_X$ is
denoted by $P_X^{kTWOWAYTAPE}$.
The original tape model corresponds to a FIFO information flow with the additional properties of rereadability and prohibition of interspersed reading and writing. Two-way tapes add the possibility of mixed FIFO and LIFO access with the same two restrictions. Once again it is found that adding this type of tape to \( P \) schemata gives no power increase. A construction which is based on ideas used in Paterson and Hewitt's discussion of the fact that every linearly recursive schema can be computed by a schema in \( P \) [Pat70] is used to show this result.

As an aid to understanding the method used in the subsequent proof, the following example, illustrating the approach, is presented. \( F \) is a functional defined by

\[
F(x,p,f,g) = g(x, g(f(x), g(f^2(x), \ldots, g(f^{j-1}(x), f^j(x))\ldots))),
\]

where \( j \) is the smallest integer \( \geq 1 \)

such that \( p(f^j(x)) = \text{true} \) \( f^0(x) \equiv x \).

\[
= x, \text{ if } j = 0.
\]

\[
= \text{undefined, if no such } j \text{ exists.}
\]

This is equivalent to the recursive formulation

\[
h(x) = \text{if } p(x) \text{ then } x \text{ else } g(x, h(x));
\]

At first blush it might seem that such a functional is not computable in \( P \) because of the necessity to "back up" in computing the \( g \) composition. However, the functional can also be expressed as
\text{J} + 0; \text{Z} + x;
\text{UNTIL p(Z) DO}
\text{BEGIN } \text{Z} + f(Z); \text{J} + J + 1 \text{ END;}
\text{W} + Z;
\text{FOR K} = J - 1 \text{ STEP } -1 \text{ UNTIL 0 DO}
\text{BEGIN } \text{Y} + x;
\text{FOR L} = 1 \text{ STEP } 1 \text{ UNTIL K DO } \text{Y} + f(Y);
\text{W} + g(Y,W)
\text{END;}
\text{HALT(W)}

and it becomes clearer that the principal problem lies in the counting performed here by J. The difficulty can be overcome, however, in the following way:

\text{Z} + x;
\text{UNTIL p(Z) DO } Z + f(Z);
\text{W} + Z; \text{IF } p(x) \text{ THEN HALT(x);}
\text{V1} + f(x);
\text{UNTIL p(V1) DO}
\text{BEGIN V2} + V1; V3 + x;
\text{UNTIL p(V2) DO}
\text{BEGIN V3} + f(V3); V2 + f(V2) \text{ END;}
\text{W} + g(V3,W);
\text{V1} + f(V1)
\text{END;}
\text{HALT(g(x,W));}

It can be seen that the number of operations (actually, applications of f) before a true value for p(f^j(x)) occurs is the number needed by the counter J and furthermore than an index based on that phenomenon can easily be decremented in the required manner.

(3.2) \textbf{Theorem: } \text{P} \text{1TWOWAYTAPE} \equiv \text{P}.

\textbf{Proof } As a starting point in creating a scheme S' in \text{P} equivalent to S in \text{P1TWOWAYTAPE}, essentially the same construction as employed in Theorem 3.1 is used. A
finite number of configurations of a one-tape schema can be "remembered" by simply recalling the contents of all simple variables of S (in other special simple variables) and the statement being executed (by replications of the schema), if the tape contents and head location are standardized in some way. Suitable standard points include configurations in which the tape ahead of the head location is logically empty (the tape is in output mode or at the end of file mark) and in which either the head is at the front of the tape or in which the tape contents between the front and the head position are of no interest.

The simulating schema S' works with the two distinguished state configurations G, the state at the beginning of generation of a tape (this state was saved in the proof technique of Theorem 3.1), and C. C is maintained such that, at any time tape operations are in input mode, if another copy of the schema were started from C it would execute H write operations before executing a rewind operation, where H is the current head position of the tape, measured from the front of the tape.

The following discussion considers how G and C are found, used, and modified. In the simulator S', when S begins to write a tape, a buffered arrangement saves in G the configuration that existed just after the last REWIND preceding the first WRITETAPE in the tape creation (or that existed at the beginning of execution, in the case of the
first tape creation). As the tape is written, C is not used until the REWIND is encountered, at which point C is set to the configuration just before the REWIND.

When the schema shifts into input mode (with something written on the tape), each forward read causes C to be "backed up" by one WRITETAPE and each backward read causes C to be "moved forward" by one WRITETAPE, according to the definition of C (the method of carrying out these operations is explained below). In the case of a forward read, if the end of file mark is detected (using the simulation methods of Theorem 3.1), C is not changed; in the case of a backward read, if the C configuration leads immediately to a REWIND (with no intervening WRITETAPE operation) the READTAPEBACKWARD is treated as a null operation and C is again left unchanged. Tests for end of file when reading can clearly be handled by testing whether C leads immediately to a REWIND statement.

The process by which C is updated must now be described. When a backward read is issued, C is merely simulated forward to the next WRITETAPE; the configuration after that WRITETAPE then replaces C (if a rewind is encountered, the configuration C is left unchanged). Obtaining the correct element for a backward read involves interleaved execution; one copy of the schema is started from G and run until it hits a WRITETAPE; another copy of the schema is then run from C until it encounters a WRITETAPE. The steps alternate until
C strikes a REWIND, at which point the correct element is available from the G path of execution. Copies must clearly be made of C and G for later use in other backward reads.

When a forward read is issued, the simulation works the same as in Theorem 3.1. In "backing up" C, execution begins with G and alternates thereafter between C and G, each execution segment being to the next WRITETAPE breakpoint, as above. When C reaches a REWIND the configuration of the G path is taken as C'. C' and G are then run alternately until C' reaches a REWIND, at which point the configuration of the G path is taken as the new C.

The above proof sketch can easily be expanded into a complete proof by straightforward construction.

Q.E.D.

A second addition to the basic tape schema class is that of markers. The markers act basically to enhance the control structure of a class, rather than to change the inherent information flow patterns, and in many schema classes studied earlier ([Con72], [Bro72]) the addition of markers led to a jump in the power of the schema classes. However, for PTAPE, as one might suspect from the methods of earlier proofs of this chapter, markers add no power. The constraints on accessing the unbounded storage are tight enough that no gain can be obtained from the marker class.
(3.3) Theorem: $P_{\text{pTAPE}}$ with markers $\equiv P$.

Proof The proof that $P_{\text{pTAPE}} \equiv P$ (Theorem 3.1) does not depend on the presence or absence of markers in the schema, so an immediate consequence of Theorem 3.1 is that $P_{\text{pTAPE}}$ with markers $\equiv P$ with markers. That $P$ with markers $\equiv P$ follows from an easily constructive simulation which uses replicated copies of a schema to keep track of the positions of markers in simple variables. The additional details should be clear.

Q.E.D.

(3.4) Theorem: $P_{\text{pTWOWAYTAPE}}$ with markers $\equiv P$.

Proof The proof is analogous to that of Theorems 3.3 and 3.2.

Q.E.D.

The foregoing results suggest that the only benefit in attaching to a machine a single tape drive with characteristics as defined above is that computation time can occasionally be saved when a group of values can be computed at some point in the execution and then used for a considerable interval without change to either values or membership. In real-life comparisons, however, it should be noted that the above discussion has not considered a tape as an input device (cf. Chapter IV), but only as a working storage device, and furthermore, that some software/hardware systems have tape drives with additional capabilities.
3.3 Two-Tape Schemata

A consideration of the power of $P$ when augmented with two tape devices of the simplest kind (no markers, only forward read) shows that the hierarchy of tape schemata is not very rich in this direction, since $P_{2TAPE}$ is a universal class. In information flow terms, the two-tape class allows two instances of FIFO rereadable storage (with strictly separated reading and writing). It is the ability to reread and to maintain two arbitrary strings of data that confers universality; end of file markers, as in the one-tape classes, are highly constrained and can appear only on the tape and then in only one logical position.

(3.5) Theorem: $P_{2TAPE}$ is universal.

Proof The result is established by showing equivalence to the universal class $P_{(3b,0)}$. The inclusion $P_{(3b,0)} \geq P_{2TAPE}$ is proved by using the fact that $P_{(3b,0)} \equiv P_{AM}$ and an obvious and easily constructible direct simulation using an array as a tape. The opposite inclusion is also shown by a direct simulation — a schema $S'$ in $P_{2TAPE}$ is constructed to imitate the behavior of schema $S$ in $P_{(3b,0)}$.

In the customary way, the interleaved Gödelization technique is employed for storage of the simulated pds's in $S'$. In the canonical arrangement, tape $T_1$ of $S'$ contains the elements in the following way:
where $A$, $B$, and $C$ are the PDS's of $S$. It will be sufficient to show how to initialize $T_1$, and how to push, pop, and test for emptiness each of the PDS's of $S$.

Since $|A| = |B| = |C| = 0$ initially, $|T_1| = 1$ initially, so the first two statements of $S'$ are $\text{WRITE} \text{TAPE}(T_1, \text{OMEGA}); \text{REWIND}(T_1);$.

Testing PDS $j$ for emptiness is performed by scanning $T_1$ to its end while computing $z = |T_1| \mod p_j$. The PDS is empty iff $z \neq 0$.

Pushing a value onto PDS $j$ is simulated by multiplying the length of the tape by $p_j$ and then shifting the elements of PDS $j$ down three elements on the tape.

a) To multiply the length of the tape

1) Tape $T_1$ is read one cell at a time, and $p_j - 1$ cells of $\Omega$ are written onto tape $T_2$ for each cell of $T_1$.

2) $T_1$, but not $T_2$, is rewound.

3) $T_1$ is copied onto $T_2$ and both tapes are rewound.

b) To shift the elements

1) $W = (|T_2| - j) \mod 3$ is computed.

2) $W$ elements are copied from $T_2$ to $T_1$. 
3) The remainder of T2 is copied to T1, using a buffer so as to shift the \((w+4)^{th}\) element to the \((w+1)^{st}\) position, etc.; the new element is put into the \((|T1| - j + 1)^{th}\) position.

Popping pds \(j\) is carried out by shifting all its elements by three cells on the tape, and then dividing the length of the tape by \(p_j\).

a) The shift actions are analogous to those for the push, but the shift is in the opposite direction.

b) To divide the length of the tape (assuming the elements are on T2 after the shift step)

1) \(|T2|/p_j\) is computed as the number of cells on T1 by writing one cell on T1 for every \(p_j\) cells of T2.

2) T1 and T2 are rewound.

3) T1 and T2 are read alternately (into dummy locations) until the end of T1 is reached. During the step \(p_j - 1\) cells of T2 are read for each cell of T1 read.

4) T1 is rewound and the remainder of T2 is copied thereupon.

Q.E.D.
3.4 **Tape Schemata with the Write-After-Read Capability**

One could object that the above models of tape units do not accurately portray conventional drives in that writing after reading is not permitted. That restriction can be relaxed in the following way:

At any time when in input mode a tape can shift directly into output mode by executing a `WRITETAPE(T,v)`. When in output mode, a direct shift to input mode can be made only by executing a `READTAPEBACKWARD(T,v)`, in which case an end of file mark is appended to the tape before the backward read (which then picks up the last-written data record) takes place. The end of file mark is appended only when the tape is in output mode just before the backward read. `READTAPE(T,v)` commands are ignored in output mode. A `REWIND` instruction causes the end of file mark to be written (if the tape is in output mode), the tape head to be positioned to the first cell, and neutral mode to be entered, as before.

The addition of this capability to the class $P_{\text{TAEPE}}$ is denoted by $P_{\text{TAEPEWITHWRITEAFTERREAD}}$ or $P_{\text{TAEPEWWAR}}$.

The write after read capability corresponds to relaxing the restrictions separating reading and writing of information on tapes. The classes formed by such augmentation have not been exhaustively investigated, but the feature, in conjunction with two-way reading, does create a more powerful class than $P$ since the effect of such additions is to create
a pushdown store whose contents can be examined without popping (a schema with one such device is equivalent to stack automata as presented in Chapter 13 of [Hop 69]).

(3.6) **Theorem:** \( P_{1\text{TWOWAYTAPEWITHWRITEAFTERREAD}} \supset P_R \)

**Proof** Section 2 of [Bro72] shows that \( P_R \equiv P(1,0) \), so it is sufficient to simulate a schema \( S \) in \( P(1,0) \) by a schema \( S' \) in \( P_{1\text{TWOWAYTAPEWITHWRITEAFTERREAD}} \). The tape is used to mimic the pd's by always keeping the tape head at the logical top of the pd's. Thus, every statement \( \text{PUSH}(PD,x) \) becomes \( \text{WRITE TAPE}(T,x) \) and every statement \( \text{POP}(PD,x) \) becomes \( \text{READ TAPE BACKWARD}(T,x) \).

Q.E.D.

From simple simulation arguments, it is also clear that superior classes to \( P_{1\text{TWOWAYTAPEWITHWRITEAFTERREAD}} \) include \( P_{2b,0} \) and \( P_{1QC}(1) \).

The situation with respect to a one-way read tape augmented by write after read capability has not been firmly determined. Such a tape cannot begin its write after read at arbitrary points, but only either at points known (and marked) during the generation of the tape or at the end of the tape (this property is apparent from a simple counting argument involving the number of distinguishable internal states of the schema). Because of the absence of a backward read, it appears to be impossible to remove the element which triggered the beginning of writing (after previously existing elements),
so that, in particular, arbitrarily many elements can be added to the end of the tape, but they can only be removed in some fixed number of chunks. For this reason it is conjectured that the class $P_{1TAPEWWAR}$ cannot simulate a pds and thus does not include the class $P_R$.

The following functional, however, can be computed in $P_{1TAPEWWAR}$:

\[ F_1(g,f,p,x) = g(y,z), \text{ where } p(y,z) = \text{true, } y = f^i(x) \text{ for } i \geq 0, z = f^j(x) \text{ for } j \geq i, \]
\[ \text{for no } k < j \]
\[ \text{and } 0 \leq m \leq k \text{ is } \]
\[ p(f^m(x), f^k(x)) = \text{true, } \]
\[ \text{for no } 0 \leq n < i \text{ is } \]
\[ p(f^n(x), f^j(x)) = \text{true.} \]
\[ = \text{undefined, if no such } y \text{ and } z \text{ exist.} \]

This is essentially a Herbrand universe search over monadic functions and dyadic predicates. The following $P_{1TAPEWWAR}$ schema computes $F_1$:

(x):  WRITETAPE(T,x); Z + f(x);
TOP:  REWIND(T);
UNTIL ENDFILE(T) DO
BEGIN  READTAPE(T,Y);
IF p(Y,Z) THEN HALT(g(Y,Z))
END;
IF p(Z,Z) THEN HALT(g(Z,Z));
WRITETAPE(T,Z); Z + f(Z);
GO TO TOP;

Implicitly, computation of $F_1$ requires two arbitrary counts
(which are kept as the head position and the tape length above) such that, essentially, one count is copied to the other, which is then decremented to zero (the counts can thus be arbitrarily different). In a P_R schema, however, the only usable method of retaining counts is by using the depth of recursion; hence it is conjectured that F1 cannot be calculated in P_R. A similar argument implies that F1 can also not be calculated in P_{n,0}.

A third intuitive argument concerns Leafest and the class P_{1TWOWAYTAPEWWAR}; it is conjectured that because on a single tape a "place" away from the head cannot be kept, and because Leafest (cf. [Pat70]) requires investigation of paths arbitrarily far apart, it is impossible to compute Leafest in P_{1TWOWAYTAPEWWAR} (that is, a schema to compute Leafest would not be able to find a spot to store the descendants it generated from a node and then return to consider the next old node; this process is essential in order to avoid being trapped in one branch of the Leafest tree).

These conjectures and their consequents may be summarized as follows:

1) P_{1TAPEWWAR} and P_R overlap, with neither being a subclass of the other [because of F1].

2) P_R < P_{1TWOWAYTAPEWWAR} < P_{1OQ(1)} [because of Theorem 3.6, F1, and Leafest].

3) P_{1TWOWAYTAPEWWAR} and P_{n,0} overlap, with neither being a subclass of the other [because of Leafest and F1].
It is surprising to find that schemata with a forward-read write-after-read tape and a single bottom-testable pds are universal because of the limited nature, when alone, of each component. As discussed above, such a tape can function as a counter which can be set to zero (by a \textsc{Rewind} and a \textsc{Write} followed by another \textsc{Rewind}), copied, or incremented, but not decremented. The ability to view a data string more than once (given by the tape) and the power to reverse an arbitrary string (given by the pds and the tape working together) seem to be the keys to the universality of the combination.

(3.7) \textbf{Theorem:} \( P_{1\text{TapeWriteAfterRead}} + (1b,0) \equiv P_{(3b,0)}. \)

\textbf{Proof} \quad The proof is accomplished by showing how to simulate three pds's with emptiness tests by the use of an interleaved Gödelization. The encoded form of the three push-down stores of a schema \( S \) in \( P_{(3b,0)} \)

\[ a_1 b_1 c_1 a_2 b_2 c_2 \ldots \Omega \]

is canonically kept in the pds of the simulating schema \( S' \) with the \( a_1 b_1 c_1 \) end at the top of the pds. As before, it is sufficient to show how \( S' \) simulates emptiness tests pushes, and pops to establish the theorem.

\textbf{Emptiness tests:} 1) The \( S' \) pds PD is copied to the tape \( T \), and \( z = |PD| \mod p_j \) is computed in the process. As earlier, simulated pds \( j \) is empty
iff \( z \neq 0 \).

2) To restore the encoded form to the pds, \( T \) is copied to \( PD \), so that the encoding is again in \( PD \) but is inverted. Another copying from \( PD \) to \( T \) to \( PD \) then restores the correct sequence.

Pushes: 1) To multiply the length of the encoded form \( PD \) is copied to \( T \), and the tape is then reread so that \( |T| \cdot (p_j - 1) \) can be computed as a length in \( PD \). The write after read feature is then used to append the contents of \( PD \) to the tape.

2) Elements are then percolated downward as necessary by a buffered copy from the tape to the pds.

3) A copying from \( PD \) to \( T \) to \( PD \) then restores the canonical configuration.

Pops: 1) If the simulated pds to be popped is empty, nothing is done; otherwise, the contents of \( PD \) are transferred to \( T \), using a buffered copy to perform the necessary percolation.

2) \( W = \frac{|T|}{p_j} \) is computed as a length in the pds \( PD \), and the tape head is moved \( W \) spaces from the beginning of the tape.

3) An end of file mark is placed on the tape by writing a dummy record and then rewinding.

4) The \( T \) to \( PD \) to \( T \) to \( PD \) copy sequence (dropping the dummy record at the first copy, say) then restores the canonical form.

Q.E.D.
It should be clear that the addition of the write after read feature, backward reads, and/or markers to the class \( P_{2TAPE} \) (or the class \( P_{kTAPE} \) for \( k \geq 2 \)) results in classes which are still equivalent to \( P_{(3b,0)} \) by simple direct constructions.

To summarize, tape-augmented schemata provide models of devices which have been in use for a number of years and give some indication of the features and adjuncts important for assuring that such devices can be effectively used. From this standpoint the result that a single tape of the two-way-read-with-markers type is not useful in increasing the power of a "raw" machine is rather unsettling, but the result that two very simple tape devices give universal power shows that the tape device has inherent capabilities (rereadability, for example) which can in certain cases of interconnection prove important.

From a schema viewpoint, the most surprising outgrowth is the conclusion that a jump from finite to unbounded storage under "reasonable" access methods does not always imply that schema power increases. Furthermore, the usual ability of markers to improve power is ineffective in this instance.

Finally, from an information flow angle, the importance of the property of persistence (rereadability) of information is illuminated; two simple tapes with essentially only this property beyond the ability to record an arbitrary string of data and afterward reread it sequentially are universal.
CHAPTER 4. PROGRAM SCHEMATA WITH UNBOUNDED INPUT AND OUTPUT

4.1 Definition of the Arbitrary Length Input Feature

The schema models developed in [Con72], [Bro72], and other sources have generally portrayed a schema as being a process which can be executed (under a given interpretation) with a fixed finite number of domain elements as input. A more realistic model of actual computing allows an unbounded number of domain elements to be input to a schema. These elements may be accessed using a GET capacity or using (in suitable cases where sufficient storage and means for detecting the end of input exist) an initialized portion of the data structure of the schema. Complementarily, a finite but unbounded sequential output stream can be modeled by PUT statements or by taking the occupied cells in some part of the schema's data structures at termination as the output of the computation.

Schemata with GETs model programs with something like card readers as input devices (where the next element is available on command, no storage is taken by the input before reading, and the input is not rereadable). PUTs model printers or card punches closely, since an item output cannot afterward be recalled and need no longer occupy space in program storage. The use of initialized data structures and finalized data structures represents systems employing writable tapes or disks as I/O devices.

Extension of schema classes in these ways essentially
provide the means for introducing one arbitrary integer
(namely, the number of elements in the input) into the calcu-
loration and for accepting an arbitrary integer as output. The
basic flow patterns are unchanged; the investigation is essen-
tially which integer functions can be performed by the various
classes.

Studies of the relationships induced by these types of
input and output lead to a large number of configurations,
many of which have somewhat unexpected positions in the power
hierarchy. The hierarchy has more levels than have thus far
been discovered in the finite input case, though many of the
distinctions break down when markers are given to every class.

The following notations are used for schema classes $P_x$
augmented with unbounded input and output, respectively:
$P_{Gx}$, $P_{P_x}$, where $G$ stands for the input function GET and
$P$ for the output function PUT. The input and output are
both regarded as sequential nonrewindable data sets (globally
accessible to all parts of a schema) which are accessed by
GET(v) and PUT(v). GET(v) places the next input element
into variable $v$ unless the entire input stream has been
read, in which case GET behaves as a null operation; PUT(v)
copies the content of $v$ into the next position of the output
data set. The capability of detecting the end of input is
given by the predicate ENDINPUT, which changes value, from
false to true, at most once (ENDINPUT is initially true if
the input stream is null) during the course of reading the
input. The ENDINPUT predicate thus acts something like a special case of a k-position switch which can be stepped (in one direction only) at externally defined times.

In definitions of functionals depending on the unbounded input feature, input elements are denoted by \( x_1, x_2, \ldots, x_n \), and the n-element ordered vector of domain elements constituting the input is represented as \( \vec{D} \) in the functional definitions (as in \( f(g,h,p,\vec{D}) \)).

Some of the results obtained for unbounded input are summarized in the following Venn diagram, together with names of functionals representative of the regions. Detailed definitions of the functionals and proofs follow the diagram.
Region | Functional
--- | ---
A | Parity($f, g, \overline{D}$)
E | Doublereverse($f, g, \overline{D}$)
G | Partialdoublereverse($f, g, p, \overline{D}$)
I | Reverse($f, \overline{D}$)
K | Any functional in P
M | if p(x1) and not p(x2) then LeafBest(P, L, R, x3)
N | Span($f, \overline{D}$)
O | LeafBest(P, L, R, x)

Regions B, C, D, F, H, J, and L are conjectured empty.

Figure 4.1 - Partial summary of results for unbounded input.
4.2 **Array, Pds, and Recursion Schemata Augmented with Unbounded Input**

In the following discussion the relationships shown in Figure 4.1 are established before the positions of queue and tape schemata in the hierarchy are studied. $P_A$, which with finite input is ineffectively universal, becomes strictly less powerful than $P_{GAe}$ when augmented with GET statements. This loss of power stems from the fact that $P_{GA}$ can input an arbitrary integer $n$ but can ascertain its magnitude only during the read-in, whereas $P_{GAe}$ can input $n$ and can determine its size more than once. The proof that $P_{GA} < P_{GAe}$ introduces a functional Reverse which is intended to highlight this difference.

(4.1) **Theorem:** $P_{GA} < P_{GAe}$.

**Proof** For input values $x_1, x_2, \ldots, x_n$ the following functional can be defined:

$$\text{Reverse}(f, D) = f(x_1, f(x_2, f(\ldots, f(x_{n-1}, x_n)\ldots)))$$

if $n > 1$

= undefined, otherwise.

Reverse, the computation of a composition of a reversed arbitrary string, can be done in $P_{GAe}$ by the following schema, but, as shown below, cannot be computed by any $P_{GA}$ schema.
HIGH ← 0; UNTIL ENDINPUT DO
    BEGIN GET(A[HIGH]); HIGH ← HIGH + 1 END;
    IF HIGH ≡ 0 OR HIGH ≡ 1 THEN HALT(OMEGA);
    Y ← f(A[HIGH - 1], A[HIGH]);
    HIGH ← HIGH ≡ 2;
    UNTIL HIGH ≡ 0 DO
    BEGIN Y ← f(A[HIGH], Y); HIGH ← HIGH ≡ 1 END;
    HALT(Y);

(The extensions ≡, OR, etc. can be simply derived from constructs strictly in the language of $P_{GAe}$.)

Reverse employs no basic predicates, so an equivalent schema $S'$ in $P_{GA}$ must be able to operate using arbitrary predicates, which is the same as autonomous operation. Theorem 9.5 of [Con72] shows that for $P_A$ schemata it is decidable whether autonomous behavior is finite or infinite. Because the predicate ENDINPUT changes only once, two copies of $S'$ (one for the false state and one for the true state of ENDINPUT) provide a $P_A$ schema (with enhanced connection between the parts) to which the reasoning in the theorem proof applies at once. The decidability of autonomous behavior follows from the fact that there is guaranteed to be a time (namely, after ENDINPUT becomes true) after which only $p + 1$ labels must be checked in order to make the decision.

If the autonomous behavior (of the second part of $S'$) is of length $k$ then there are infinitely many inputs of length $k + 1, ...$ for which Reverse cannot be calculated (because they cannot even be reread and the composition requires that the elements be examined in reverse order). If
the autonomous behavior is infinite, on the other hand, \( S' \) will never halt (and thus cannot compute Reverse). Therefore, \( S' \) cannot exist.

Q.E.D.

It should be clear from the proof and from general considerations that markers \( P_{\text{GAM}} \) added to the array class give sufficient power to compute Reverse, and furthermore, that the same proof methods used to show \( P_{\text{AM}} \equiv P_{\text{Ae}} \) establish that \( P_{\text{GAM}} \equiv P_{\text{GAe}} \).

In spite of the decrease in relative power suffered by the simplest array schemata, the class \( P_{\text{GA}} \) is still superior to the class \( P_{G(n,0)} \), since arrays can still simulate markerless pds behavior under unbounded input. The simulation is easy, with the exception of the special case of popping an empty pds. The direct proof is exhibited below because in this case, as in several others, the locator-type proof used in an earlier work is difficult to patch up for the unbounded input case; the reason for the problem is that an arbitrary amount of input may have to be saved, together with a detectable marker indicating the end of the input, and this latter capability is not effectively available in \( P_{\text{GA}} \).

\[
(4.2) \quad \text{Theorem: } \ P_{\text{GA}} > P_{G(n,0)} .
\]

\textbf{Proof} \quad \text{The relation } P_{\text{GA}} > P_{G(n,0)} \text{ follows at once from a direct simulation in which an array is used to represent}
each pds of the $P_{G(n,0)}$ schema $S$. To handle the case of a pop of an empty pds, which must be a null operation (one leaving the state of the schema unchanged), parallel arrays of pointers and elements are kept according to the following method. If a particular pds contains (top to bottom) three elements $a$, $b$, and $c$, they are stored as

<table>
<thead>
<tr>
<th>Cell index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELT</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTR</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

$\text{TOP} = 1; \quad \text{BOTTOM} = 4; \quad \text{NEXT} = 5$

The indices stored in the array $\text{PTR}$ give the successor element to the one currently being accessed in the parallel array $\text{ELT}$. The simple variables $\text{TOP}$, $\text{BOTTOM}$, and $\text{NEXT}$ indicate the top of the pds, the next-to-the-bottom element of the pds, and the next free element of the array, respectively. A $\text{PUSH(PD,v)}$ statement then becomes

\[
\begin{align*}
\text{ELT}[\text{NEXT}] & \leftarrow v; & \text{The value is inserted.} \\
\text{PTR}[\text{NEXT}] & \leftarrow \text{TOP}; & \text{The successor pointer is set.} \\
\text{TOP} & \leftarrow \text{NEXT}; & \text{The index of the pds top is adjusted.} \\
\text{NEXT} & \leftarrow \text{NEXT} + 1; & \text{The free space indicator is updated.}
\end{align*}
\]

and a $\text{POP(PD,v)}$ instruction becomes

\[
\begin{align*}
\text{ELT}[\text{BOTTOM}] & \leftarrow v; & \text{The old value of } v \text{ is put into array in case } PD \text{ is empty,} \\
v & \leftarrow \text{ELT}[\text{TOP}]; & \text{and a new value is fetched.} \\
\text{TOP} & \leftarrow \text{PTR}[\text{TOP}]; & \text{The index of the pds top is adjusted.}
\end{align*}
\]

The above technique insures that a pop of an empty pds leaves
the value of the target variable unchanged.

The strict inclusion follows from the computability of $\text{LeafTest}(P,L,R,x)$ in $P_{GA}$ and not in $P_G(n,0)$.

Q.E.D.

Region $N$ of Figure 4.1 includes functionals that are not computable in either $P_{GA}$ or $P_G(1,m)$ (the functional Reverse employed in Theorem 4.1 is clearly computable in $P_G(1,m)$, since schemata in that class can examine an input integer twice). A functional in that region is, for input $x_1, x_2, \ldots, x_n$, Span:

\[
\text{Span}(f, D) = f(x_{n/2}, f(x_{n/2} + 1, f(\ldots, f(x_2, f(x_{n-1}, f(x_1, x_n)))\ldots))) \quad \text{for} \quad n = 2k \text{ and} \quad k = 1, 2, 3, \ldots.
\]

= undefined, otherwise.

Span is defined only for positive even-length inputs and is formed (without dependence on basic predicates) by composition over the input string in a manner which takes elements alternately from the end and beginning of the string.

(4.3) Theorem: Span is computable in $P_{GAe}$ but in neither $P_{GA}$ nor $P_G(1,m)$.

Proof The obvious algorithm for computation of Span in $P_{GAe}$ will not be detailed here (Span can in fact be computed in $P_G(2b,0)$).

The intuitive reason why Span is not computable in $P_{CA}$
is the same as that for the noncomputability of Reverse in $P_{GA}$. Once the input string has been read, a schema $S$ in $P_{GA}$ can no longer determine the length of that string, and the nature of the Span functional requires that a) the entire input be seen before calculation is begun, and b) an amount of calculation dependent on the input length (and therefore unencodable in the schema itself) be done.

If $S$ is a $k$-variable $P_{G(1,m)}$ schema to compute Span, then under autonomous behavior (implied by the Span definition) and with ENDINPUT held at false, $S$ loops before executing more than a fixed number $N$ statements (the proof of this is a direct analogue to the reasoning in Section 2 of [Bro72]). Furthermore, by earlier results ([Bro72], Section 2) only a fixed number of variables is required to simulate the behavior of $S$ under strict autonomous behavior. The definition of Span requires that all input be read and saved before computation begins. Thus, after $N$ values have been read, at least one value $x_j$, $j < N$, has either been discarded (which would violate the requirements of Span) or has been pushed onto the pds so far that it cannot be retrieved until a predicate changes value. The only test available is ENDINPUT, which may not change value until arbitrarily many values have been input. In the computation of Span, the value $x_j$ mentioned above will be required after a fixed amount of calculation ($2 \cdot j - 2$ function applications). Then, however, arbitrarily many
(n - 2\cdot j + 1) values of the input have not yet been employed in the calculation and must still be saved. To reach the value $x_j$, all but a fixed number of these elements must be removed from the $pds$ and kept. But there are only $k$ simple variables. Hence $S$ cannot exist.

The two above outlines can be formalized to complete the proof.

Q.E.D.

The functional Reverse has been shown to lie outside the region enclosed by $P_{GA}$ in Figure 4.1, and it is necessary to determine where among the regions shown the functional does lie. The following lemma establishes directly that Reverse is computable in one of the simplest classes.

(4.4) **Lemma:** Reverse is computable in $P_{GR}$.

**Proof** The following source code shows how to accomplish the desired end.

```plaintext
IF ENDINPUT THEN HALT(OMEGA);
GET(X);
IF ENDINPUT THEN HALT(OMEGA);
HALT(f(X,FUNCT()));

FUNCT(): BEGIN GET(Z);
        IF ENDINPUT THEN HALT(Z)
        ELSE HALT(f(Z,FUNCT()))
END
```

The execution-time activation record stack is used to keep the input until it can be reexamined (by exiting recursive
calls), and the implicit detection of the end of the input when it is being read backward (because of the fact that a HALT in the outermost recursion level stops execution) allows Reverse to be computed correctly.

Q.E.D.

This lemma is directly useful in showing the relationship between $P_{GRgM}$ and $P_{G(1,0)}$, as the following proof demonstrates.

(4.5) **Theorem:** $P_{GRgM} > P_{G(1,0)}$.

**Proof** The simulation of $S$ in $P_{G(1,0)}$ by $S'$ in $P_{GRgM}$ is essentially identical to that in Lemma 2.2 of [Bro72]. Global variables are used for the simple variables of $S$, markers are used (in other global variables) to indicate the desired point to which control should be transferred after a return, and the activation record stack of $S'$ is used to keep the elements of the pds of $S$ (by keeping the pds elements in local variables, they are automatically stacked at each recursive call).

Since $P_{G(n,0)} \geq P_{G(1,0)}$, Theorem 4.2 and the proof of Theorem 4.1 show that Reverse is not computable in $P_{G(1,0)}$. The strict inclusion to be shown here follows at once from Lemma 4.4.

Q.E.D.

Not all finite-input equivalences are upset by the
addition of unbounded input, however, for the classes \( P_{RgM} \) and \( P_{(1,m)} \) remain equivalent under such augmentation. The internal structure and operation of the two classes is very similar, though, so that the result might be intuitively expected.

(4.6) **Theorem:** \( P_{GRgM} \equiv P_{G(1,m)} \).

**Proof** The inclusion \( P_{GRgM} \supseteq P_{G(1,m)} \) follows at once from Lemma 2.2 of [Bro72] by simple extension.

To show that \( P_{G(1,m)} \supseteq P_{GRgM} \), a simulator \( S' \) is constructed for a schema \( S \) in \( P_{GRgM} \). The method is, without loss of generality, for the case in which \( S \) has but one recursively called function \( F \) with a single parameter. The following changes are made to a copy of \( S \) to form \( S' \):

1) All local variables \( v_1, \ldots, v_m \), and the formal parameter \( w \) within \( F \) are renamed to be different from any names outside of \( F \).

2) Each one of the \( t \) statements of \( S \) is assigned a label \( l_j \) which is keyed to its ordinal position in the schema. The label of the first statement of \( F \) is \( l_F \).

3) The statement \( \text{PUSH}(PD,M) \) is placed at the front of \( S' \).

4) Each call to the function \( F \) is constrained to be a simple assignment statement of the form \( X \leftarrow F(Y) \), and each such assignment is replaced by
Z = F(Y); X = Z, for a new simple variable Z.

5) Each statement $\ell_j$: Z = F(Y) is replaced by the block

$$
\ell_j: \begin{align*}
\text{BEGIN} & \quad \text{PUSH(PD,M);} \\
& \quad \text{PUSH(PD,$v_1$);} \\
& \quad \text{$v_1$ = OMEGA;} \\
& \quad \vdots \\
& \quad \text{PUSH(PD,$v_m$);} \\
& \quad \text{$v_m$ = OMEGA;} \\
& \quad \text{PUSH(PD,w);} \\
& \quad \text{w = OMEGA;} \\
& \quad \text{PUSH(PD,OMEGA); $\leftarrow$ repeated j times} \\
& \quad \vdots \\
& \quad \text{PUSH(PD,M);} \\
& \quad \text{PUSH(PD,OMEGA); $\leftarrow$ repeated t - j - 1 times} \\
& \quad \vdots \\
& \quad \text{PUSH(PD,OMEGA);} \\
& \quad \text{GO TO $\ell_f$;}
\end{align*}
\text{END}
$$

A separator goes in.
The values of local variables and the formal parameter are initialized.
The return point is marked.
The function is entered.

6) Each statement $\ell_j$: HALT(Y) is replaced by the block

$$
\ell_j: \begin{align*}
\text{BEGIN} & \quad \text{POP(PD,U);} \\
& \quad \text{IF U = M THEN} \\
& \quad \quad \text{HALT(Y);} \\
& \quad \text{POP(PD,$u_1$);} \\
& \quad \vdots \\
& \quad \text{POP(PD,$u_t$);} \\
& \quad \text{POP(PD,w);} \\
& \quad \text{POP(PD,$v_m$);} \\
& \quad \vdots \\
& \quad \text{POP(PD,$v_1$);} \\
\end{align*}
$$

If no activation record is present, the top recursion level has been reached.
The return label is temporarily saved.
The formal parameter and the local variables are reset.
POP(PD,U); The separator is removed.
Z + Y; The value of the invocation is returned.

IF U1 = M THEN
  GO TO $\lambda_1$
ELSE IF U2 = M A branch is made to the return point.
  THEN GO TO $\lambda_2$
  .
ELSE GO TO $\lambda_t$

END

Q.E.D.

An interesting consequence of addition of unbounded input to schemata is the separation of $P_G(1,m)$ and $P_{GRg}$. One reason for this separation is the inability of $P_{GRg}$ to pass a control state change back through levels of recursive calls (that is, a return from a function must always be to the statement following the call, regardless of what takes place at deeper levels of recursion; $P_G(1,m)$, however, can use its markers to branch away from that statement, depending on deeper-level calculation). The problem does not arise for the finite input case because of the existence of locators which yield the markers necessary for returning the control state, but with unbounded input such locators do not exist (and markers are thus effective in adding power).

(4.7) **Theorem:** $P_G(1,m) > P_{GRg}$.

**Proof** For arbitrary input $x_1, \ldots, x_n$, the functional Parity can be considered:
Parity\( (f, g, \overline{D}) = f(x_1, f(x_2, f(\ldots, f(x_{n-1}, x_n)\ldots))) \)
for \( n = 2 \cdot k \) and \( k = 1, 2, 3, \ldots \)
\( = g(x_1, g(x_2, g(\ldots, g(x_{n-1}, x_n)\ldots))) \)
for \( n = 2 \cdot k + 1 \) and \( k = 1, 2, 3, \ldots \)
\( = \text{undefined, otherwise.} \)

Suppose \( S \), which has \( v \) simple variables (counting both local and global variables), is a \( P_{GRg} \) schema to compute Parity. Then \( S \) can be changed to an equivalent schema \( S' \) in \( P_{GRg} \) in which all actual parameters are simple variables and all schema function calls are part of simple assignments: \( x + h_1(y) \). In a nonbasic function \( h_2 \) when statement \( j \) of the form \( x + h_1(y) \), where \( h_1 \) is nonbasic, is executed, control information is stacked so that whenever \( h_1 \) returns (executes a \( \text{HALT}(z) \)) the value it returns is stored into \( x \) and control is transferred to statement \( j+1 \) in \( h_2 \).

The definition of Parity requires that unboundedly many input values be read and saved before computation is begun. Consequently, after \( v \) values have been read, additional storage must have been generated by the only possible means, namely, recursive calls, which give additional instances of local variables. There are no basic predicates in the specification of Parity and no markers available, so the only deviation from autonomous behavior in \( S \) occurs when \( \text{ENDINPUT} \) changes from \( \text{false} \) to \( \text{true} \).

By Theorem 3.11 of [Con72], consequently, no "unwinding" of the recursion can take place until \( \text{ENDINPUT} \) changes value.
At the time the last element is read (and not until then) it can be determined whether $n$ is even or odd, and therefore whether the $f$ or the $g$ composition should be the result of Parity. However, as indicated, all but a fixed number (less than $v$) of elements of the input are implicitly stored in the recursion stack, so the invocation in which ENDINPUT changes must be exited while some computation remains to be done.

Return from a recursive call promptly causes control to pass to the statement following the call, and autonomous behavior that is independent of the parity of $n$ (because ENDINPUT always changes value exactly once, from false to true, irrespective of the parity of $n$) must subsequently be followed. Therefore, the same path to a HALT (if the schema loops, Parity certainly cannot be computed) must always be followed in higher levels (including the outermost block), regardless of the behavior of $S'$ in the invocation in which ENDINPUT changed. But these higher levels must perform computation that is part of the output and must select a value to be the result of $S'$. But that computation will be invariant over different-length inputs, in contradiction to the specification of Parity. Hence $S'$ (and therefore $S$) cannot exist.

Q.E.D.

(4.8) Corollary: Parity is not computable in $P_G(n,0)$. 
Proof. The corollary follows directly from the noncomputability of Reverse in $P_G(n,0)$.

A related result is the strict inclusion of $P_{GR}$ within $P_{GRg}$. This phenomenon is partly due to the restriction that schema-level functions return only one value, according to definition. Although this distinction may seem like mere quibbling over the way in which $P_R$ is set up, it is worth noting that it states that the addition of global variables does in some cases increase the power of a class.

\[(4.9) \textbf{Theorem:} \quad P_{GRg} \supset P_{GR}.\]

Proof. The relation $P_{GRg} \supset P_{GR}$ is immediate. Strict containment follows because of the below functional:

$$\text{Doublereverse}(f,g,\overline{D})$$

$$= f(f(x_1, f(x_2, f(\ldots, f(x_{n-1}, x_n), \ldots))),$$

$$g(x_1, g(x_2, g(\ldots, g(x_{n-1}, x_n), \ldots)))$$

for $n > 1$;

$$= \text{undefined, otherwise.}$$

Doublereverse requires simply that Reverse be computed twice (once with $f$'s and once with $g$'s), and then that the two results be combined by an application of $f$.

A simple modification to the $P_{GR}$ schema to compute Reverse allows computation of Doublereverse in $P_{GRg}$. Only an outline of why Doublereverse cannot be computed in $P_{GR}$
will be given. Basically, the proof revolves around the inability of a $P_{GR}$ schema to pass more than one value back up through its stack of recursive invocations.

Suppose $S$ in $P_{GR}$ is a schema with $v$ simple variables which computes Doublereverse. As in the proof of Theorem 4.7, after a fixed number of input elements have been read, some of the input value must have been stored on the implicit stack of recursive calls (Doublereverse, like Parity, requires that all input be read before computation is begun). At some invocation $i$, ENDINPUT changes value and the calculation can begin.

Invocation $i$ can employ in its computation at most the input values $x_{n-v+1}, \ldots, x_n$; a return must be executed to obtain earlier values in the sequence. The $f$ and $g$ compositions, which at both intermediate and final stages are domain elements, must be kept separate until the last step of the calculation. However, the definition of schema functions ([Con72], Definitions 2.1 and 3.1) allows only one value to be returned from a call. Combination of the two partially completed compositions at the return from invocation $i$ cannot be done since two domain elements cannot be encoded into a single domain value. On the other hand, if invocation $i$ is exited to allow one of the compositions to be further computed, the input values stored in the local variables of $i$ are irretrievably lost, and the second composition can
thus never be performed. Hence $S$ cannot exist.

Q.E.D.

(4.10) Corollary: Doublereverse($f,g,D$) is not computable in $P_g(n,0)$.

The onion-like strict inclusions that generally hold among classes with finite inputs do not usually remain when arbitrarily many inputs are allowed; this phenomenon is true because if classes $P_X$ and $P_Y$ are related by $P_X < P_Y$, it is frequently the case that schemata in $P_{G_X}$ can examine the magnitude of the number of input elements twice, whereas those in $P_{G_Y}$ cannot. That dichotomy leads at once to an overlap relation, as the following three proofs show.

In the first case, the implicit detection of the end of the input when rereading (as discussed above in Lemma 4.4) permits schemata in $P_{GR}$ to measure $n$ twice, whereas schemata in $P_{G(n,0)}$ can examine $n$ only once.

(4.11) Theorem: $P_{GR}$ and $P_{G(n,0)}$ overlap, with neither being a subclass of the other.

Proof: In a Venn diagram keyed to Figure 4.1, the relationship of the theorem appears as
It is sufficient for the proof to display functionals falling in the regions $M$, $K$ and $I$.

$M$: By the same type of proof used to show $P_R < P(n,0)$ (Theorem 3.5 of [Bro72]) the functional

$$F(P,L,R,x,y,z) = \text{if } P(x) \text{ and not } P(y) \text{ then}$$

$$\text{Leafest}(P,L,R,z) \text{ else } \Omega$$

cannot be computed in $P_{GR}$ but can be computed in $P_{G(n,0)}$.

$K$: Any functional in $P$ can be computed by a schema in $P_{GR}$ and also by a schema in $P_{G(n,0)}$.

$I$: By Theorems 4.1 and 4.2 the functional Reverse cannot be computed in $P_{G(n,0)}$, but by Lemma 4.4 Reverse is computable in $P_{GR}$.

Q.E.D.

(4.12) Corollary: $P_{G(1,m)}$ and $P_{G(n,0)}$ overlap, with neither being a subclass of the other.
Proof. Only the additional fact that Reverse can be computed in $P_{G(1,m)}$ need be proved to establish the corollary. The following schema shows how Reverse can be done:

\begin{verbatim}
PUSH(PD,M);
UNTIL ENDINPUT DO
    BEGIN GET(Z); PUSH(PD,Z) END;
POP(PD,Y); POP(PD,X);
IF X = M OR Y = M THEN HALT(OMEGA);
Y = f(X,Y);
POP(PD,X);
UNTIL X = M DO
    BEGIN Y = f(X,Y); POP(PD,X) END;
HALT(Y);
\end{verbatim}

Q.E.D.

(4.13) Theorem: $P_{G(1,0)}$ and $P_{GR}$ overlap, with neither being a subclass of the other.

Proof. Again by using a Venn diagram (as in Theorem 4.11)

\begin{center}
\begin{tikzpicture}
    \node (PG) {$P_{G(1,0)}$};
    \node (GR) at (3,0) {$P_{GR}$};
    \node (G) at (1,0) {G};
    \node (K) at (1.5,0) {K};
    \node (I) at (2,0) {I};
    \draw (PG) -- (G) -- (K) -- (I) -- (GR);
\end{tikzpicture}
\end{center}

the following results are obtained:
G: For input $x_1, x_2, \ldots, x_n$ the functional Partialdoublereverse is defined to be

$$\text{Partialdoublereverse}(f,g,p,D)$$

$$= f(g(x_k,g(x_{k+1},g(\ldots,g(x_{n-1},x_n)\ldots))),$$

$$f(x_k,f(x_{k+1},f(\ldots,f(x_{n-1},x_n)\ldots))),$$

where $n > 1$ and $p(x_k) = \text{true}$ for some $k \leq n - 1$ and not $p(x_i)$ for all $i = 1, 2, \ldots, k - 1$

= undefined, otherwise.

Intuitively, this functional searches for the first input element $x_k$ which gives $p(x_k) = \text{true}$, and then, considering $x_k$ as the beginning of the input string with which it is concerned, computes the Doublereverse functional of Theorem 4.9.

The markers that the true predicate provides clearly permit a schema $S$ in $P_{G(1,0)}$ to calculate Doublereverse in the same way that $P_{G(1,m)}$ does, but such markers still do not permit a schema in $P_{GR}$ to pass back more than one value from recursive calls (cf. Theorem 4.9). The computation cannot be repeated, locator-style, because the input cannot be retained.

K: Any functional in $P$ is contained in zone $K$. 
I: Reverse has been shown computable in $P_{GR}$ (Lemma 4.4) and not computable in $P_{G(1,0)}$ (Theorem 4.5).

Q.E.D.

4.3 Application of Unbounded Input to the Class $P_{(2b,0)}$

The question of the position of $P_{G(2b,0)}$ in the hierarchy is uncertain, as it is for the finite input case. Certainly there exist functionals which imply that $P_{G(2b,0)}$ is not a subclass of $P_{GA}$, as the following lemma shows.

(4.14) Lemma: There exists a functional computable in $P_{G(2b,0)}$, but not in $P_{GA}$ or $P_{G(1,m)}$.

Proof The earlier introduced functional $\text{Span}(f,\bar{D})$ (Theorem 4.3) is such a functional. It can be computed in $P_{G(2b,0)}$ by the following simple schema:

```
IF ENDINPUT THEN HALT(OMEGA);
UNTIL ENDINPUT DO
    BEGIN GET(X); PUSH(PD1,X);
            IF ENDINPUT THEN HALT(OMEGA);
            GET(X); PUSH(PD1,X)
    END;
POP(PD1,X);

LOOP2: POP(PD1,Y);
UNTIL EMPTYPDS(PD1) DO
    BEGIN PUSH(PD2,Y); POP(PD1,Y) END;
    X + f(Y,X);
    IF EMPTYPDS(PD2) THEN HALT(X);
    POP(PD2,Y);
UNTIL EMPTYPDS(PD2) DO
    BEGIN PUSH(PD1,Y); POP(PD2,Y) END;
    X + f(Y,X);
    GO TO LOOP2;
```

Q.E.D.
An intuitive estimate of how the markers available in $P_G(1,m)$ affect the relationship of that class to $P_G(2b,0)$ is difficult, since the markers could well combine with the $pds$ capability in some way to allow $P_G(1,m)$ to compute a functional not in $P_G(2b,0)$ just as happens with Reverse in the case of $P_{GA}$. The following rather tedious proof shows that this is not so, however, and, together with Theorem 4.16, verifies that $P_G(2b,0)$ is a superior class to $P_G(1,m)$. The relationship between $P_G(2b,0)$ and $P_{GA}$ probably cannot be answered until the connection between $P_A$ and $P(2b,0)$ is settled.

An interesting feature about the proof of Lemma 4.15 is that it follows essentially a locator and $p$-simulator plan as has been earlier used in [Con72] and [Bro72], but that in this instance the locator operates in a class considerably more powerful than $P$.

(4.15) Lemma: $P_G(1,m) \subseteq P_G(2b,0)$.

Proof. For an arbitrary schema $S$ in $P_G(1,m)$ a locator and $p$-simulator schema $S'$ in $P_G(2b,0)$ is constructed. $S'$ uses one of its $pds$'s $PD1$ to save the input for use when (if) a true predicate is found (without loss of generality, all predicates are assumed to yield $false$ when their arguments are $OMEGA$; the full form of the locator is discussed in [Con72]). The $p$-simulator that results is
conceptually simple; first, the remainder of the input (if any) is read onto PD1, and PD1 is poured into PD2 so that the elements are accessible in the proper order. PD1 is then used to simulate the pd's PD of S, and the value for which a predicate is true is used to simulate the markers, as described in Theorem 3.6.

The locator section of S' uses its pd's PD2 to store elements of PD, and employs an elaborate reverse execution technique to keep up with the positions of markers. Without loss of generality, the succeeding discussion assumes there is only one marker (cf. [Con72], Theorem 5.3 and 5.4).

The autonomous execution of S may be thought of as having two parts—the section performed before the end of input is reached, and the part executed after ENDINPUT changes value. To separate these two conditions the locator is formed from two copies of S which are first modified and interconnected in the following way:

a) Each copy is completely labeled in the same sequential fashion, so that the statements of the first copy are $t_{l_1}$ and of the second, $t_{2_j}$.

b) Every statement $t_{l_j}: \text{GET}(v)$ in the first copy is replaced by $t_{l_j}: \text{BEGIN GET}(v); \text{IF ENDINPUT THEN GO TO } t_{2_j+1} \text{ END }$. 
c) Every statement IF ENDPINPUT THEN S1 ELSE S2
    (except those added in step b) is replaced by S2
    in the first copy and S1 in the second copy.

Now under autonomous behavior, if ENDPINPUT is considered to
be uniformly false, a repeated configuration will occur in
the first copy within a fixed number of statement executions
(this is the basis of the locator for $P_{1,m}$), and the net
number $k$ of elements placed on the PDS before the loop is
entered can be readily discovered. Furthermore, the net
number of elements $m$ placed on the PDS during each exe-
cution of the loop can be calculated, and the sequence of
domain elements and markers can be determined in each case.
The format of the PDS when ENDPINPUT becomes true within
the loop will thus be of the form

$$e_{11} e_{12} \cdots e_{1k} (e_{21} e_{22} \cdots e_{2m})^* e_{21} \cdots e_{2j}$$

bottom of the PDS

where each $e_{xy}$ is either a domain element or a marker.

Logically, the second copy of $S$ is now replaced by many
different copies of $S$, each one to remember a particular
point in the above sequence where ENDPINPUT can become true,
and the copies are linked to the first copy by changing the
GO TO statements inserted after the GETS above. The locator
then works in these new copies of $S$ by moving backward
through the $e_{xy}$ sequence as net pops are executed from PD
(it should be kept in mind that this part of the locator is executed only after ENDINPUT becomes true). By keeping track of which variables contain markers (thus requiring more replications of S) and noting when markers are popped from the pds (by examining the $e_{xy}$ sequence), the action of S can be simulated.

Two rough points remain to be dealt with: first, the movement backward through the $e_{xy}$ sequence depends on net pops; clearly if S both pushes and pops its pds after ENDINPUT changes value, some additional bookkeeping must be done. Added copies of S can be used to determine how many elements have been placed on PD on top of the last pre-ENDINPUT element; if more than a fixed number, which is roughly, for m markers and v variables in S,

$$|S| \cdot (m + 1) + (v + 1),$$

are placed there, S is in a loop (which will not be exited until a predicate changes value) and the old elements (including the remaining ones inserted before the end of the input) will never be popped.

The second problem is to determine how to know when to exit the starred portion of the $e_{xy}$ sequence when moving backward. A buffer used between the simulating pds PD2 and the logical pds PD allows the schema to tell when exactly k elements remain on PD. At that point the starred
portion is exited and the sequence \( e_1, \ldots, e_{11} \) is followed to determine which popped values are markers.

The detailed construction of the hydra-like simulator \( S' \) follows straightforwardly from the above description.

Q.E.D.

\[(4.16) \textbf{Theorem:} \quad P_G(1,m) \prec P_G(2b,0). \]

\textbf{Proof} \quad The theorem is a direct consequence of Lemma 4.15 and the noncomputability of Leafest in \( P_G(1,m) \) (which follows from the fact that Leafest cannot be computed in \( P(1,m) \)) and from the recognition that unbounded input can add no computational power because no program-writable storage or program-governable control features are added).

Q.E.D.

The question of whether \( P_{G\ Ae} \) is more powerful than \( P_G(2b,0) \) may be answered when the relation between \( P_{Ae} \) and \( P(2b,0) \) is solved, but it seems that regardless of that outcome, \( P_G(2b,0) \) is strictly included in \( P_{G\ Ae} \). The basis for such a conjecture is the existence of the functional Lasthalf:

\[
\text{Lasthalf}(f,\overline{D}) = f(x_n, f(x_{n-1}, f(\ldots, f(x_{k+1}, x_k), \ldots)))
\]

\[\text{for } n = 2 \cdot k \text{ and } k = 1, 2, \ldots\]

\[= \text{undefined, otherwise.}\]
It seems to be impossible for a schema in $P_G(2b,0)$ to find the middle point of an arbitrary input string without discarding at least the last half of the string in the process. Last half is, of course, easily computable in $P_G\lambda e$.

4.4 Queues and Unbounded Input

The foregoing theorems deal principally with the schema classes introduced by Constable and Gries and other earlier workers. The interrelationship among such classes is quite complex in itself, so the study of tape and queue schemata has been separated into the following discussion.

As might be expected, markers also play an important role in determining inclusions in queue classes, since such classes have sufficient information flow capabilities to be universal when the ability to mark one or two places in the data structure is given. Absence of markers, however, implies that queues can only examine the number of elements once (though they can examine the elements themselves many times), in contrast to one-pds and recursive classes. The following lemma indicates how the proof of such a statement goes.

(4.17) Lemma: The functional Reverse is not computable in $P_G\lambda Q(0)$.

Proof As in earlier proofs, Reverse must be computable without the use of basic predicates, by its definition. Thus,
if $S$ is a schema in $P_{GlQ(0)}$ which computes $\text{Reverse}$, consideration of the autonomous behavior of $S$ leads to the following conclusion:

If $\text{ENDINPUT}$ is fixed at false and $S$ in that instance halts in a finite number of statements (which must be $\leq |S|$) or enters a loop which does not test $\text{ENDINPUT}$, $S$ cannot compute $\text{Reverse}$ since there exist infinitely many cases in which it cannot even read all of the input. Otherwise, if the behavior of $S$ for $\text{ENDINPUT} = \text{false}$ is traced and $\text{ENDINPUT}$ is then allowed to become true, either $S$ will halt in $\leq |S|$ further statements or $S$ will loop. In both cases, it is clear that $S$ cannot compute $\text{Reverse}$.

The above outline can be easily expanded to a rigorous proof.

Q.E.D.

\[(4.18) \textbf{Corollary: } \text{Reverse is not computable in } P_{GnQ(0)}.\]

The surprisingly powerful class $P_{G(1,m)}$ can be shown by simple arguments to overlap with the class $P_{GlQ(0)}$, principally because of the markers it possesses.

\[(4.19) \textbf{Theorem: } P_{G(1,m)} \text{ and } P_{GlQ(0)} \text{ overlap, with neither being a subclass of the other.}\]
Proof. As in earlier arguments, the theorem is established by displaying an element of each of the regions I, K, and O of the diagram below:

\[ \text{PG}_{1,\text{m}} \quad \text{I} \quad \text{K} \quad \text{O} \quad \text{PG}_{\text{GLQ}(0)} \]

I: Reverse has been shown computable in \( \text{PG}_{1,\text{m}} \) and not computable in \( \text{PG}_{\text{GLQ}(0)} \).

K: Any functional computable in \( P \) can be computed in each of the above classes.

O: \( \text{PG}_{\text{GLQ}(0)} \) can compute \( \text{Leaftest}(P,L,R,x) \), whereas \( \text{PG}_{1,\text{m}} \) cannot (as before, the inability of \( \text{PG}_{1,\text{m}} \) to compute \( \text{Leaftest} \) follows at once from the inability of \( P_{(1,\text{m})} \) to compute that functional and the easily demonstrable fact that the addition of unbounded input does not affect that proof).

Q.E.D.

The same sort of argument as above can be extended to show how \( \text{PG}_{\text{GLQ}(0)} \) relates to several other classes.
(4.20) **Theorem:** \( P_{GLQ(0)} \prec P_{GLQ(1)} \).

**Proof**  
Lemma 2.9 can be extended to show that \( P_{GLQ(0)} \leq P_{GLQ(1)} \). In \( P_{GLQ(1)} \) the functional Reverse can be computed by the following schema, which uses the C-type queue \( Q \) and the chip \( C \):

\[
\text{IF ENDINPUT THEN HALT(OMEGA);} \\
\text{GET}(X); \\
\text{IF ENDINPUT THEN HALT(OMEGA);} \\
\text{UNTIL ENDINPUT DO} \\
\text{BEGIN PUSHLEFT}(Q,X); \text{GET}(X) \text{ END;} \\
PUSHLEFT(Q,C); \text{POPRIGHT}(Q,Y); \\
\text{UNTIL } Y = C \text{ DO} \\
\text{BEGIN POPRIGHT}(Q,Z); \\
\text{UNTIL } Z = C \text{ DO} \\
\text{BEGIN PUSHLEFT}(Q,Y); \\
Y + Z; \\
\text{POPRIGHT}(Q,Z) \\
\text{END;} \\
X + f(Y,X); \\
PUSHLEFT(Q,Z); \\
\text{POPRIGHT}(Q,Y) \\
\text{END;} \\
\text{HALT}(X); \\
\]

However, by Lemma 4.17, Reverse is not computable in \( P_{GLQ(0)} \), so the strict inclusion is established.  

**Q.E.D.**

(4.21) **Corollary:** \( P_{GnQ(0)} \not\equiv P_{GLQ(1)} \).

(4.22) **Corollary:** \( P_{GLQ(0)} \prec P_{G(2b,0)} \).

**Proof**  
The corollary follows at once from Theorem 2.7 (extended slightly) and Theorem 4.20.
The above theorems, together with slightly broadened earlier results, such as Theorem 2.7, justify the following diagram:

![Diagram](image)

Figure 4.2 - Queue classes with unbounded input

It should be clear that the universal queue formulations under finite input \((P_{1Q(2)}, P_{2Q(1)}, \text{etc.})\) are equivalent to \(P_{\text{GAe}}\) when unbounded input is added.

4.5 **Tapes and Unbounded Input**

The input of an arbitrary integer causes a real splitting among classes of tape schemata with respect to power, for the different types of single tapes are able to bring their different information flows to bear on the length of the input so as to compute several sets of functions. For instance, the persistence characteristic of the simplest one-tape schemata allows the following functional to be calculated for input \(x_1, \ldots, x_n:\)

\[
\text{Rightshift}(f, \overline{D}) = f(x_{n-1}, f(x_{n-2}, f(\ldots, f(x_1, x_n), \ldots)))
\]

for \(n > 1\)

= undefined, otherwise.
The definition of Rightshift requires that all the input be kept until the last element is read, and then that a modified forward composition under $f$ be performed. That Rightshift can be computed in $P_{GlTape}$ is demonstrated below:

(4.23) **Lemma:** $P_{GlTape}$ can do Rightshift.

**Proof** The following schema computes the functional:

```
IF ENDINPUT THEN HALT(OMEGA);
GET(X);
IF ENDINPUT THEN HALT(OMEGA);
UNTIL ENDINPUT DO
  BEGIN
    GET(Y);
    WRITETAPE(T,X);
    X ← Y
  END;
REWIND(T);
READTAPE(T,Y);
X ← f(Y,X);
UNTIL ENDFILE(T) DO
  BEGIN
    READTAPE(T,Y);
    X ← f(Y,X)
  END;
HALT(X);
```

The cases for which Rightshift is undefined are checked.
The input is written onto the tape, with a buffered arrangement always keeping the last value separated.
The tape is reread and the composition computed.

Q.E.D.

Schema configurations whose information flow is LIFO only, such as one-pds and recursion classes, are unable to compute Rightshift, however, as this outline shows:

(4.24) **Lemma:** $P_{G(1,m)}$ cannot compute Rightshift.

**Proof** The definition of Rightshift requires

1) all input must be read before any computation
germane to the function can be carried out;

2) the input must be considered in the order
in which it was read (except for the last element).

The only way a schema $S$ in $P_{G(1,m)}$ can store an arbitrary number of domain elements is by inserting them into its pds.

When $S$ is operating autonomously and ENDINPUT is fixed at either `true` or `false`, $S$ can execute no more than (for $S$ having $m$ markers and $v$ simple variables)

$$T = |S| \cdot (m + 1) + v$$ statements before it begins to loop.

When $S$ is reading the input (ENDINPUT = `false`) and a new value $x_r$ is read, $x_r$ must therefore be placed either in a simple variable or within $T$ statements of the top of the pds.

$S$ can reach down only $t < T$ positions into the pds, so when $r$ becomes $r_0 > t + v$ there will be elements $x_j$, $j \leq r$, in the pds that are saved nowhere else. These elements will eventually have arbitrarily many $x_i$, where $i > r_0$, pushed on top of them (because $n$ can be arbitrarily large, and only a fixed number of values can be kept in the simple variables). When ENDINPUT changes value and the composition is computed, all $x_j$, $j \leq r_0$, must be accessed before the $x_i$, $i > r_0$, can be used. But this requires popping an arbitrary number of values and keeping them in only a fixed number of variables, which is impossible.

Q.E.D.
The functional Reverse, however, which is computable in classes such as $P_{G(1,m)}$ and $P_{GR}$, cannot be computed in a single-tape configuration with only forward reads.

(4.25) **Lemma:** Reverse cannot be computed in $P_{GLTAPENWAR}$ with markers.

**Proof** Reverse is a functional which by its definition uses no basic predicates and requires that the entire input be seen and saved before computation is begun. If $S$ is a schema in $P_{GLTAPENWAR}$ with markers which claims to compute Reverse, the autonomous behavior of $S$ can be considered. As $S$, which has $v$ simple variables, reads input values $x_i$, after at most $v$ inputs are read, an input element must be written on the tape or all values cannot be retained. Eventually, as in Lemma 4.24, elements will have to be written on the tape within a fixed distance of the end at the time of writing (if the write after read feature is used to add elements at points other than the end, all input elements following the addition must be saved temporarily, and there are only $v$ locations which can be used for such a process). This implies that arbitrarily many elements are arbitrarily far out of order (with respect to a forward read) when the composition must be computed.

Now using the write-after-read feature at the end of the tape permits only additions (and not deletions) there. Deletions from the tape can be achieved only by writing after
some marked point a shorter string than the string that was there before. A (buffered) read to the end of the tape, on the other hand, can extract only a fixed number of elements, which remain written on the tape. The only other possibility for obtaining the arbitrarily many elements near the end of the tape is to use the markers to govern deletions of entire blocks of elements from the end of the tape as the computation proceeds.

For $S$ to fetch values, therefore, it must read down its tape to some marked block, fetch at most $v$ values, compute the composition, rewind its tape, scan down to a block previous to the one just read, etc., in order to overcome the arbitrary separation problem. However, in order to store $N$ input variables, where $N$ is large (approximately $n$), $\sim N / v = L$ blocks are needed. But this implies that $S$ has to pass as many as $L$ markers, where $L$ is arbitrarily large, before stopping. But the number of internal states of $S$ is fixed (less than $|S| \cdot (m + 1) + v$), so it cannot pass arbitrarily many markers without getting into a loop from which it cannot emerge. Thus the functional cannot be computed.

The above intuitive proof can be suitably formalized to establish the result.

Q.E.D.

Another interesting result is that the addition of
unbounded input causes the class of single two-way write-after-read tape schemata to be strictly larger than that of one-way tapes and that of one pdps with markers.

(4.26) Lemma: There exists a functional which can be computed in $P_{GLTOWAYTAPEWWAR}$ but not in $P_{G(1,m)}$ or $P_{GLTAPEWWAR}$.

Proof The functional Nestedreverse is such a computation:

$$\text{Nestedreverse}(f, \bar{D}) = f(x_1, f(x_2, f(..., f(x_n, f(x_1, f(x_2, \\
\hspace{20pt} f(..., f(x_{n-1}, x_n)...)))...)))$$

for $n > 1$

= undefined, otherwise.

Nestedreverse cannot be computed in $P_{GLTAPEWWAR}$ because Reverse is contained as a subfunctional, and Lemma 4.25 states that Reverse is not computable in $P_{GLTAPEWWAR}$. Nestedreverse cannot be computed in $P_{G(1,m)}$ because it requires viewing arbitrarily many elements twice in the same order. Calculation of Nestedreverse cannot begin until all input values have been read (by its definition), and the computation requires that all values be examined one time, and then examined again. Because $S$ in $P_{G(1,m)}$ must store the elements in order in its pdps and must pop its pdps to use the elements in the composition, the popped elements must be saved outside the pdps for use in the function's second half.
But since there is only a fixed number of simple variables, not all values can be saved. Formalization of this argument along the lines of the proof of Lemma 4.24 concludes the proof.

Q.E.D.

As do other classes that are effectively universal under finite input $P_{2TAPE}$ retains its universality when unbounded input is added, as a simple examination of Theorem 3.5 shows.

The above lemmas have established the following Venn diagram which shows graphically how the power hierarchy changes with the addition of unbounded input. The large number of possible comparisons is responsible for the fact that so many relationships, both among tape classes and between tape classes and other structure classes, are as yet uninvestigated.
Region | Functional
--- | ---
A | Reverse
B | Any functional in P
C | Rightshift
D | Nestedreverse
E | Leaf test (conjectured)

Figure 4.3 - Tape classes with unbounded input

4.6 **Implications of Unbounded Output**

Some of the preceding material has been concerned with the effects of adding an unbounded (but finite) input capacity to program schemata. An obvious extension is the addition of an unbounded output by means of a special function $\text{PUT}(x)$, which places $\text{content}(x)$ into the next position of the infinite serial write-only nonrewindable output medium (the parameter of a $\text{HALT}(y)$ statement is appended to the output just before the schema is stopped).

Two of the points connected with the unbounded output augmentation are the necessity to reconsider locator-type
proofs and the possible production of output by a nonterminating schema. With respect to the first matter, locators can be modified in the following way:

1) All PUT statements are deleted from the locator.
2) All HALT statements are replaced by a branch to the first statement of a "new" copy (with different variable names, labels, etc. and with properly initialized input variables) of the schema for which the locator is written.

With respect to the second point, it should be noted that although it would be possible for all schemata to be nonterminating and proper results still to be produced, it would be impossible to tell when a schema had run long enough. Therefore, for simplicity, the output medium is viewed as a blind storage device which is not examined until the schema executes a terminating HALT statement.

From the above modifications of locators and from an examination of the proofs, it can be seen that for $P_{PX}$ schemata the power hierarchy is no different from that for $P_X$ schemata.

A further topic which has not been well developed, but which brings the unbounded input/output models even closer to reality, is the introduction of initialized and finalized data structures. A part of a schema's data structures can be initialized to contain the elements of an arbitrarily long
input stream, and a portion of the data structure can be required to contain the output of the computation. Clearly, the initialized input is of interest only in the cases where a schema class has sufficient capability to be able to detect the end of the stream. In both initialized and finalized classes, an ordering of the data to be input or output, as well as a place in accessible storage, must be specified.

The interest in such a topic arises from its indication of how the resulting restriction on working space affects the class of functionals computable by a schema group, since in most real situations the space used for input and output could also be used as working space for the program. As can be seen at once, the restrictions have an effect only when the number of structures is limited (e.g., having as many pds's as desired effectively allows a schema to circumvent directly any space cramping problems) and when accessing is limited in some as-yet-not-understood way. The implications of the effects on middle-range classes are of most interest (for instance, $P_{I(4b,0)}$ is clearly universal, but $P_{I(3b,0)}$ is conjectured not to be). Only a modicum of study has been devoted to these classes, which are largely left for future research.

The effect of adding an unbounded input is therefore seen to be a significant one, since classes that are in the case of finite input simply related often spread apart into an overlapping or more complex relationship. The significance
of such results is that the finite input does not adequately permit the inherent power of an information flow pattern to be expressed and studied. Furthermore, the unbounded input, as mentioned, more closely approximates actual program conditions. Clearly, an additional direction that is suggested by the research is the introduction of multiple inputs (corresponding to inputting more than one arbitrary integer) and input streams with other characteristics than sequencibility and nonrewindability (such as rereadability, which gets around the problem of ascertaining the magnitude of the input more than once, as pointed out in the proofs, but which does not overcome the data flow inadequacies highlighted by the inability of certain classes to compute, for example, Reverse).
CHAPTER 5. INTERCONNECTION OF MACHINES

5.1 Definition of the Machine Interface

It should be clear by this point that the concept of a program schema models computers, or machines (with the basic functions and predicates playing the part of black boxes in the hardware), in the same way that it models programs. Recent advances in microprogramming research have demonstrated how tenuous the distinction between hardware and software is, and this blurring is to some degree reflected in schema studies. Furthermore, the approach to schemata through the classes of functions they compute is closely related to the approach of theory of computation researchers to finite automata, pushdown automata, Turing machines, etc. through the classes of languages they accept.

Frequently in the study of classes of program schemata a new class $C'$ is created by augmenting a class $C$ with some feature such as a pushdown store, natural number arithmetic, or markers. It seems reasonable that under certain conditions $C'$ can also be regarded as being a class defined by machines of classes $C_1, C_2, \ldots, C_k$ communicating through a finite memory. That is, it is intuitively expected that the class $P(2b,0)$, for instance, be the same as the class of interconnected $P(1b,0)$ machine pairs:
Consideration of such interconnection is a rudimentary step toward studying schema formulations for parallel processing, but it should also be noted that the two connected machines can have a definite master-slave relationship as well as the more commonly considered separate-but-equal status. The investigation is also important for determining the properties that a machine composed of connected submachines will have. The following paragraphs define the mechanisms of connection and communication between two processors S' and S" which are joined together.

A finite length FIFO communications queue is associated with each processor, and to the list of schema instructions are added two transmission commands SEND(\(\overline{v}\)) (\(\overline{v}\) denoting a list of values) and RECEIVE(v) which, respectively, place values onto and remove a value from a message queue. If processors S' and S" are connected, then in processor S', for instance, execution of a SEND transmits as an indivisible operation one or more variable values, in order, to the message
queue of processor \textit{S"}. The variables to be transmitted are listed explicitly in the call, as a varying length parameter list might be, so the maximum number of values which can be sent at once is fixed for a given schema. Execution of a \texttt{RECEIVE(v)} in \textit{S'} removes the oldest value from the message queue of \textit{S'} and places it into the variable \texttt{v} (\texttt{v} is set to \texttt{Ω} if the queue is empty). The \texttt{RECEIVE} command operates on only one value so that messages of different lengths (from different \texttt{SEND} commands) can be handled conveniently. Removing one element at a time and testing the queue for emptiness then permits information to be passed as the length of the message (but only finitely many codes can be transmitted because of the requirement that the parameter list to a \texttt{SEND} be explicitly stated).

The test \texttt{IF EMPTYMESSAGEQUEUE THEN S1 ELSE S2} checks a processor's own message queue for emptiness; the predicate is automatically set to \texttt{true} when the last element of a message is read and is locked from changing until either it is tested or a \texttt{RECEIVE} is performed, thus avoiding some synchronization problems. A further restriction is that only one message (or part thereof) can be on a message queue at a time. \texttt{SEND} commands executed while the target \texttt{EMPTYMESSAGEQUEUE} predicate is locked cause the sending process to wait until the queue is unlocked as above. Deadlock is thus a possibility. The two processors can communicate only by means of the message queue; variable names, labels,
etc. are local to each processor configuration.

Certain reasonable assumptions are also made about the way processors operate. In particular, both processors are always executing instructions except when blocked as above. Secondly, there is a fixed number \( N \) of instruction classes into exactly one of which every instruction falls, such that every command in class \( t_i \) has a fixed execution time \( T_i \). \( T_i \) is expressed as an integral positive multiple of some basic time unit \( z \). Exact time conflicts between operations (between a SEND of processor \( S' \) and a RECEIVE of processor \( S'' \), say) are hardware-resolved in favor of \( S' \). This hardware preference removes symmetry between parallel processes in many cases. Finally, the output of a composite schema is defined to be the output of the first subschema that halts.

An additional restriction must be placed on the behavior of basic functions \( f_i \) and predicates \( p_i \). The \( f_i \) and \( p_i \) are considered to be total functions, as in earlier works, and, because exact timing is crucial in determining schema behavior under parallelism, computation time of these functions is specified. If the basic function and predicate times are allowed to be interpretation dependent, it would seem that an undesirable dependence is injected, and in such a way that simulation of a parallel schema by a serial schema becomes impossible. For example, the parallel P-schema
(x): \( Y + f(x); \)  
HALT(Y);  

(x): \( Z + g(x); \)  
HALT(Z);

operating under the premise that the first member which halts gives the schema output, cannot be simulated in \( P_{AE} \), since that class has no way to determine which of \( f \) and \( g \) takes longer to execute. Parenthetically, it should be noted that the existence of an external program-readable clock or timer would permit such a determination. Furthermore, under variable basic operation times the conventional mapping from the free interpretation to other interpretations no longer holds ([Pat67]).

Consequently, as in Karp and Miller [Karp69] and other studies, basic operations are taken to have a constant time requirement, and for simplicity, that requirement is assumed to be one time unit.

The formulation of communications behavior is probably close to reality with the possible exception of the limitation of the message queues to single messages. Two reasons exist for this decision: a) the original goal was to consider communication through a finite control, and a multiple message queue would effectively mean adding the new schema feature of an unbounded buffer store, and b) if the message queue were unboundedly long, then two processors with markers (either actual markers or domain elements and a predicate) could use the message queues for integer arithmetic, which would be an unintended and undesired use of the communications facility.
As an example, if the message queue is permitted to carry multiple messages, two schemata in $P$ with markers have the power to perform integer arithmetic by using identical copies of a schema in $P$ as the communicating processes. The integer values are kept in Gödelized form on the message queue, and the schemata are forced to execute alternately. Markers are used to specify the next statement to be executed and to demarcate the ends of the Gödelization. A detailed proof of how such a configuration can simulate a schema in $P$ is given in the Appendix as Lemma 5.1.

5.2 Restrictions on the Processors

One of the principal goals of this study is to establish requirements on the structures of processors $S'$ and $S''$ so that their union is equivalent to the composite processor $S$. The processors can be considered as representatives of schema classes $P_A$, $P_B$, and $P_C$. The connection of schematopes $P_A$ and $P_B$ can be denoted by $P_A \times B$ (which, because of the hardware priority mentioned above, is not the same as $P_B \times A$).

In the following paragraphs only certain schema features are considered: arrays, counters, markers, pds's, queues, and tapes. For simplicity, the presence of arrays in a class specification implies the use of only one array since multiple arrays can be simulated in a single array. (Theorem 4.4 of [Con72]).
The following two restrictions then are shown by Theorems 5.2 and 5.3 below to be sufficient conditions for the desired connectivity relations to hold. Necessary requirements for connectivity have yet to be established.

R1: If a counter, a pads, a queue, or a tape is considered as a feature, then $P_A$ and $P_B$ taken together must permit exactly the same number and type of features as $P_C$.

R2: Markers or arrays may be included in at least one of $P_A$ and $P_B$ if and only if the corresponding feature is allowed in $P_C$. Whenever markers are included in either $P_A$ or $P_B$ they may be passed across the communications link.

(5.2) **Theorem:** Given classes $P_A$, $P_B$, and $P_C$ which satisfy R1 and R2, then $P_C \preceq P_A \times B$.

**Proof** The basic idea of the proof is to create a master-slave relationship between $S'$ in $P_A$ and $S''$ in $P_B$ and to pass control information back and forth by encoding based on the number of elements in each message. If arrays are included in either $P_A$ or $P_B$, the controlling schema is chosen to be the one which has the highest array power ($P_{Ae}$ over $P_A$, for example); otherwise, $S'$ is designated as the controlling, or master, schema. "Master" in this usage
refers to a control superiority and not to the hardware tie-breaking priority mentioned earlier.

The master schema, say $S'$, is built, then, by beginning with a copy of $S$ in $P_C$ and replacing each manipulation of an RI-type element found only in $S''$ (such as a particular pdS) by a block containing code to

1) send an encoded message specifying the manipulation to be performed;
2) wait until a reply arrives in the message queue of $S'$;
3) receive the returned data or acknowledgement of command;
4) send data, if necessary (such as an element to be pushed);
5) loop until a reply or acknowledgement arrives;
6) receive the acknowledgement;
7) act on the result, if necessary.

The basic skeleton can, of course, be varied if no data transmission from $S'$ to $S''$ is necessary (as in the case of an emptiness test) to a simple send-receive pair. A counter which executes RECEIVES and tests the message queue for emptiness is used to classify the response.

For units of the slave schema, codes must (potentially) be assigned for actions of each feature of each type as below:
<table>
<thead>
<tr>
<th>Feature</th>
<th>Send Code (sent by $S'$)</th>
<th>Reply Code (sent by $S''$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tape</td>
<td>Rewind</td>
<td>Command executed</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Command ignored (for</td>
</tr>
<tr>
<td></td>
<td>Read backward</td>
<td>read past end of file, etc.)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test end of file</td>
<td>End of file true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End of file false</td>
</tr>
<tr>
<td>Queue</td>
<td>Popleft</td>
<td>Command executed</td>
</tr>
<tr>
<td></td>
<td>Popright</td>
<td>Command ignored</td>
</tr>
<tr>
<td></td>
<td>Pushleft</td>
<td>Emptiness true</td>
</tr>
<tr>
<td></td>
<td>Pushright</td>
<td>Emptiness false</td>
</tr>
<tr>
<td></td>
<td>Test emptiness</td>
<td></td>
</tr>
<tr>
<td>Pds</td>
<td>Pop</td>
<td>Command executed</td>
</tr>
<tr>
<td></td>
<td>Push</td>
<td>Command ignored</td>
</tr>
<tr>
<td></td>
<td>Test emptiness</td>
<td>Emptiness true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Emptiness false</td>
</tr>
<tr>
<td>Counter</td>
<td>Increment</td>
<td>Command executed</td>
</tr>
<tr>
<td></td>
<td>Decrement</td>
<td>Command ignored</td>
</tr>
<tr>
<td></td>
<td>Test for zero</td>
<td>Zero test true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zero test false</td>
</tr>
<tr>
<td>Marker</td>
<td>Test value for marker</td>
<td>Value is this marker</td>
</tr>
<tr>
<td></td>
<td>Return copy of marker</td>
<td>Value is not this marker</td>
</tr>
</tbody>
</table>

Figure 5.1 - Action codes used in interschema communication

Clearly these codes can be represented by a fixed number of integers for a given $S$ in $P_C$.

$S''$ is constructed to loop waiting for a command to arrive, and then to branch on the code received to the appropriate service routine. When the command has been completely processed (including the second exchange of messages, if necessary) $S''$ returns to its initial looping state.

The above outline can be fleshed out to a full proof of the containment $P_C \leq P_A \times B$ under the given restrictions.

Q.E.D.
(5.3) **Theorem:** Given classes $P_A$, $P_B$, and $P_C$ satisfying the restrictions $R_1$ and $R_2$ above, then $P_C \supset P_A \times B$.

**Proof** To establish the theorem a schema $S$ in $P_C$ must be constructed to model the (time dependent) behavior of schemata $S'$ in $P_A$ and $S''$ in $P_B$. $P_A$, $P_B$, and $P_C$ must meet the requirements $R_1$ and $R_2$ listed above, so all features usable in either $P_A$ or $P_B$ are available to $S$ in $P_C$.

The bases for the simulation are a) simulation of the communications link, and b) simulation of the execution synchronization of the processes. The maximum length $k$ of messages that can be sent can be determined by inspection of the parameter lists of the SEND commands, so $k$ new simple variables can be allocated to hold the values of the message elements. Copies of the schema being built can be replicated to keep track of the length of the message queue of each coprocessor and whether each queue is locked from accepting values, as above; details of such bookkeeping are shown below. The outcome of emptiness tests and the effect of executing SEND instructions is then known for each such subsection of the total simulating schema.

With respect to the second aim of the simulation, it can be seen that time dependencies are important only as they affect the sending of messages back and forth. An interpretation independent means of simulating such messages must be
devised.

The concept of a control point is useful in what follows. Basically, a control point is a division of a statement into parts whose execution time can always be determined on a schema level. If a statement can be considered as a unit which always takes a fixed time, the entire statement can be taken as a control point. On the other hand, statements like IF-THEN-ELSE and BEGIN-END have portions which have a fixed overhead and portions which vary with individual schemata, and control points are designated accordingly.

At the beginning of the simulation both processors $S'$ and $S''$ are synchronized with respect to instruction executions. Unless a processor is blocked by an attempt to SEND a message to a companion processor that is not ready, there is a fixed bound (namely, the largest instruction time) on the time span between inception of successive instructions (as stated earlier, three separate times are assigned to an IF-THEN-ELSE construction and two to a BEGIN-END group; statements within these compound units are counted separately). Thus when the two schemata $S'$ and $S''$ are running concurrently, the maximum time between the termination of instructions (where all instructions in both schemata are considered) must also be less than a fixed bound (in the absence of total deadlock).

For all unblocked situations, therefore, there are no more than a fixed number of synchronization states
(combinations of control points in each schema, message queue lengths, and time to completion for the current instruction in either schema):

$$|S'|_C \cdot |S''|_C \cdot (\max |Q'| + 1) \cdot (\max |Q''| + 1) \cdot \max T_i$$

where $|S'|_C$ and $|S''|_C$ are the numbers of control points (slightly larger than the number of statements because of the splitting of statements) in the two subschemata,

$$\max |Q'| \quad \text{and} \quad \max |Q''|$$

are the maximum number of elements in a SEND parameter vector in $S''$ and $S'$, respectively (these numbers are increased by one to allow for the empty-but-locked state), and

$$\max T_i$$

is the maximum instruction time.

For these states there exists a well-defined transition function (dependent in some cases, of course, on the outcome of basic predicate tests).

In the states for which the next instruction is a SEND and the object message queue is nonempty or locked, transitions can be to a separate set of (for a blocked SEND in $S'$)

$$|S''|_C \cdot \max |Q''| \cdot \max T_i$$

states, each of which "knows" to awaken the blocked schema if the message queue is unlocked, as well as the particular execution point and queue size of the blocked schema. A SEND which is blocked in this set of states implies that the entire parallel composition is deadlocked.
To form the synchronization states, it is apparent that many replications of statements in $S'$ and $S''$ will have to be made. Each statement in each schema can be separated out and its control points labeled ordinarily, and copies of control points can be designated by

$$S[u_{S'}, u_{S''}, \ell_S', \ell_S'', q_{S'}, q_{S''}, t_{S'}, t_{S''}, b_{S'}, b_{S''}, w]$$

where $u_R = \text{control point in schema } R$

$\ell_R = 1 \text{ if the message queue of schema } R \text{ is locked and } 0 \text{ otherwise}$

$q_R = \text{size of the message queue of } R$

$t_R = \text{time for current operation in } R$

$b_R = 1 \text{ if } R \text{ is blocked and } 0 \text{ otherwise}$

$w = 1 \text{ if the control point referenced is in } S' \text{ and } 2 \text{ if in } S''$.

As in earlier proofs, not all combinations of subscripts are utilized, but all are included to enhance the clarity of the sketch (for instance, $b_R$ must be 0 except when $u_R$ points to a SEND).

The transitions between synchronization states can now be specified. For state

$$S[u_{S'}, u_{S''}, \ell_S', \ell_S'', q_{S'}, q_{S''}, t_{S'}, t_{S''}, b_{S'}, b_{S''}, w]$$

if $b_{S'} = b_{S''} = 1$, a transition is made back to the same control state (deadlock loop);
otherwise, if \( w = 1 \) (for \( S' \)) then

a) the control point referenced, \( u_{S'} \), is executed. Some action, such as assignment to a variable, may be required.

b) A successor control point \( u'_S \), is determined by analyzing the normal control flow within the schema. In the case of an EMPTYMESSAGEQUEUE predicate, the test is true iff \( l'_S = 0 \) and \( q'_S = 0 \).

c) A new transition state with primed subscripts is determined according to the following rules:

\( u'_{S'} = \) successor to \( u_S \), chosen in step b

\( u''_{S'} = u_S \)

\[
\begin{align*}
\ell'_{S'} &= \begin{cases} 
0 & \text{if } u_{S'} \text{ is a RECEIVE or an EMPTYMESSAGEQUEUE predicate} \\
& \text{and } q_{S'} = 0 \\
\ell_S & \text{otherwise}
\end{cases}
\]

\[
\ell''_{S'} = \begin{cases} 
1 & \text{if } u_{S'} \text{ is a SEND} \\
\ell_S & \text{otherwise}
\end{cases}
\]

\[
q'_{S'} = \begin{cases} 
q_{S'} - 1 & \text{if } u_{S'} \text{ is a RECEIVE} \\
q_{S'} & \text{otherwise}
\end{cases}
\]
\[ q'_{S''} = \begin{cases} 
\text{number of elements in the SEND if } u_{S'}, \\
\text{is a SEND and } l_{S''} = 0 \\
q_{S''} \text{ otherwise}
\end{cases} \]

\[ t'_{S'} = \begin{cases} 
\infty \text{ if } u_{S'}, \text{ is a SEND and } l_{S''} = 1 \\
\text{time associated with } u'_{S'}, \text{ control point otherwise}
\end{cases} \]

\[ t'_{S''} = \begin{cases} 
\infty \text{ if } b'_{S''} = 1 \\
\text{time of a SEND if } b_{S''} = 1 \text{ and } b'_{S''} = 0 \\
t_{S''} - \text{time of a blocked SEND if } u_{S'}, \text{ is a SEND and } l_{S''} = 1 \\
t_{S''} - t_{S'}, \text{ otherwise}
\end{cases} \]

\[ b'_{S'} = \begin{cases} 
1 \text{ if } u_{S'}, \text{ is a SEND and } l_{S''} = 1 \\
b_{S'}, \text{ otherwise}
\end{cases} \]

\[ b'_{S''} = \begin{cases} 
0 \text{ if } u_{S'}, \text{ is a RECEIVE or an EMPTYMESSAGEQUEUE predicate} \\
\text{and } q_{S'} = 0 \\
b_{S''} \text{ otherwise}
\end{cases} \]
\[ w = \begin{cases} 1 & \text{if } t'_s \leq t'_s'' \\ 2 & \text{otherwise} \end{cases} \]

The transitions for the state \( w = 2 \) are similar.

Suitable formalization of the above outline concludes the proof.

Q.E.D.

Related to the above connection theorem is the outcome when numbers of equivalent machine classes are combined. Equivalence is not preserved under the connection relation, as the following example quickly shows:

\[ P(1,0) \equiv P(1,m) \quad \text{but} \quad P(1,0) \times (1,0) \not\equiv P(1,m) \times (1,m) \]

since by Theorem 5.2 the first combination is equivalent to \( P(2,0) \) and the second to \( P(2,2m) \) (a universal class). The general case of what power class results when two devices are connected arbitrarily has not been researched at all, but appears suited to an information flow analysis. Such a topic is of practical importance because of the light it may shed on problems involving connection of subunits in such diverse places as hardware design of digital computers and network interconnection of machines.
CHAPTER 6. OPEN PROBLEMS, DIRECTIONS FOR
FURTHER RESEARCH, AND CONCLUSIONS

A large part of effort expended in recent program schema
research has been in attempts to define the field, to es-
tablish directions for growth, and to investigate areas of
possible application for both the schema approach and for
specific results. For these reasons, there are large open
areas which are only marginally developed, several of which
are discussed in this chapter. Elimination of the identity
function (direct assignment) and the position of the class
P(2b,0) are discussed rather extensively, but several other
points are also made. Proofs for theorems cited are found in
the Appendix.

6.1 Schemata without the Identity Function

Program schemata as defined and used in [Con72], [Bro72],
and the foregoing pages are equipped with one feature that
seems close to being an interpreted function—the notion of
the identity assignment of the form \( x + y \), where \( x \) and
\( y \) are variables. In its most schematized form the assignment
\( a + b \) can be considered as \( a + f(b) \), where \( f \) is the
identity function.

The proof of Theorem 6.1 indicates how the removal of
the identity function from the simplest schema class \( P \) can
be accomplished without loss of power by essentially using a
mapping technique to associate groups of variables with particular cells containing data (investigation of a similar case is reported in [Gar71]). Replicated copies of the subject schema are employed to keep track of the time-varying membership in the equivalence classes (of variables) which are defined by the relationship "has been set equal to by direct assignment."

(6.1) **Theorem:** P without the identity function $\equiv$ P with the identity function.

**Proof** The proof is contained in the Appendix.

The definition of copy-free schema classes becomes somewhat more difficult for classes with data structures, since "cheating" of the following ilk must be excluded:

$$x + y; \text{ can be simulated by the sequence }$$

```
PUSH(PD,y); POP(PD,x);
```

Perhaps the most straightforward way of accomplishing such a separation is to depart from the "normal" computer analogy and to consider all information as a physical entity which is moved about (in the same way that chips move) by assignment and data structure manipulation, with the undefined element $\Omega$ filling gaps that occur. Under such an interpretation all markers degenerate to chips, where the number of chips available is determined by the number of initialized variables
(a new construct) which contain the value of a marker. Initialized variables are certain locations which are set to designated schema constants, such as chips and subscript seeds (the subscript 0) before execution of a schema begins; initialization is much like the process whereby the input variables of a finite input schema receive values before execution begins.

In addition to these general considerations, certain specifications for the behavior of arrays and subscripting when the identity function is lacking must be made before the behavior of the class \( P_{\text{Ae}} \) can be investigated.

1) The increment function ' + 1 ' for subscripts is considered not to transfer an identical value; thus the statement \( v \leftarrow w + 1 \); where \( v \) and \( w \) are subscript variables, leaves \text{content}(w) \) unchanged and sets \text{content}(v) \) to \( (\text{content}(w) + 1) \).

2) The identity assignment \( v \leftarrow w \) is prohibited for subscript values as well as for domain values.

3) All schema elements are initially undefined.

Then the following theorem can be proved in much the same way that the corresponding theorem for \( P \) was proved:
(6.2) **Theorem:** \( P_{\text{Ae}} \) **without the identity function** \( \equiv P_{\text{Ae}} \) **with the identity function.**

**Proof** The proof is contained in the Appendix.

The situation at interior points of the hierarchy is not as neat, for several reasons. One set of difficulties arises from locator-type proofs and another applies rather generally to any schema having a non-randomly accessible data storage structure. The first locator problem is that the initial values of a schema must be used by both the locator and the p-simulator and consequently must be preserved while the locator is running. This rough spot can be handled, happily, in the following way. If a schema \( S \) has \( k \) input variables \( v_1, \ldots, v_k \), its locator is constructed as before, except replicated \( 2^k \) times. The copies of the locator are designed \( S[d_1, d_2, \ldots, d_k] \), where \( d_j \) is either 1 or 0, depending on whether the \( j^{\text{th}} \) input variable has received a new value since the locator was started. In all copies in which \( d_j \) is 1 (a new value has been assigned) all instances of \( v_k \) are replaced by \( v_k' \), where the primed variables represent new locations. The copies are linked by substituting for each statement \( \ell[d_1, d_2, \ldots, d_k, t] \):

\[
v_j + \ldots ; \text{ the block}
\]

\[
\ell[d_1', \ldots, d_k', t]: \begin{align*}
&\text{BEGIN} \quad v_j' + \ldots ; \\
&\text{GO TO} \quad \ell[d_1', \ldots, d_{j-1}', 1, d_j+1', \\
&\quad \ldots, d_k', t] \\
&\text{END}
\end{align*}
\]
Therefore, no input variable ever receives a new value by assignment, and the input values remain unchanged.

The second potential difficulty lies in the possible lack of enough computed markers (copies of RT or RF) to perform the p-simulation. However, as long as RT is not an input value, by using the above technique of preserving the starting conditions of the schema, the locator can be run several times to obtain identical values in different variables (by changing the names used by the locator) without violating the restrictions implied by the absence of the identity function. The limitation on the number of markers present in a scheme (as discussed above) then allows this method to simulate all necessary markers. In certain situations, as will be discussed below, the case in which the RT value is from an input variable can be handled by adding to the code of the schema.

A difficulty that applies generally to schemata having a data structure that is not randomly accessible is that a value may be inserted into the data structure and then be used for later computation: \( \text{PUSH}(PD, W); W \leftarrow f(W) \); or a value may be inserted into a data structure several times. In such situations as these a technique must be found for duplicating computations, for introducing temporary variables, or for taking checkpoints (for instance, if the value \( W \) is pushed several times and it is known that \( W \) was created from \( f(Y) \),
then by keeping \( Y \) and \( f \) as checkpoint information, as many copies of \( W \) as needed may be pushed; popping becomes difficult, however, because of the necessity to restore the checkpoint somehow).

As an illustration of the problems involved with domain markers which are input variables, the following theorem is presented:

\[
\text{(6.3) Theorem:} \quad P(n,0) \quad \text{without the identity function} < P(n,0) \quad \text{with the identity function.}
\]

The proof of the theorem (detailed in the Appendix) rests on the fact that \( P(n,0) \) with one chip is not universal, whereas \( P(n,0) \) with one marker is universal, and the fact that an input value which yields \text{true} for some predicate behaves as a marker when the identity function is present and as a chip otherwise.

The consequences of omitting the identity function are only beginning to be studied, and though results have been shown for the weakest and strongest power classes, there are prospects of several separations in the middle of the power range, analogously to the case in which unbounded input is added. Here, however, there exist undetermined relationships not only between classes without the identity function, but also between comparable classes with and without identity assignment, so the breadth of the potential study is greatly extended.
The principal value of each study lies in its indication of just how important a single interpreted function is. It is already indicated that the identity assignment operates as a tremendously compacting shorthand in some cases (P_A^c, notably), and for that reason the operation of the feature is worth study because of its indication of similar extant or hypothetical abbreviations. Furthermore, the removal of identity assignment corresponds really to a restriction that is placed on information flow within the finite control unit rather than between the control unit and data structures or among data structures. The further study of the topic may therefore lead to a fuller understanding of the mechanisms of information flow.

6.2 The Position of the Class \( P(2b,0) \)

One of the major unresolved questions in past work which has been done on the program-like schema form introduced by Constable and Gries and developed by Brown, Gries, and Szymanski is the position in the power hierarchy of the class \( P(2b,0) \). Conjectures are that \( P(2b,0) \) is not universal, though the problem has resisted all efforts to this point. The class has the capability to perform integer arithmetic using the two \( \text{pds}'s \) and is therefore quite near to universality, but it has so far proved impossible to devise a suitable data storage method that allows both retention of
arbitrarily many domain elements in an accessible format and
performance of arithmetic at the same time. Several interest-
ing results with respect to augmentations which yield
universality have been developed, however.

The first such enhancement is with an ordinary markerless
pds. The proof (in the Appendix) demonstrates the ability of
$P(2b,0)$ to perform integer arithmetic.

(6.4) Lemma: $P(2b,0) + (1,0)$ is universal.

(6.5) Corollary: $P(3,0)$ with two chips is universal.

Perhaps a more interesting augmentation is with the
predicate TALLER(PD1,PD2), which yields true iff the height
(number of cells in) pds PD1 is greater than the height
of pds PD2. Testing for equality of pds heights is then
simply accomplished by the combination

IF NOT TALLER(PD1,PD2) AND NOT TALLER(PD2,PD1)

THEN S1 ELSE S2

The value of such a predicate is that it permits the height
of one pds to be duplicated as the number of cells in the
second pds, without disturbing the first pds; that
ability permits a simulation of the class $P(3b,0)$, as
the proof of the following lemma (contained in the Appendix)
shows.
(6.6) **Lemma:** \( P(2b,0) \) with the predicate TALLER is universal.

(6.7) **Corollary:** \( P(2b,0) \) with the predicate \( \text{EQUALHEIGHT}(PD_1,PD_2) \) is universal.

The interest in such an addition, besides the pinpointing of a weakness in \( P(2b,0) \) if the class is indeed subuniversal, is in its parallel to the position of the identity function. Each of the two predicates is an interpreted predicate (neither of which, by the way, requires more knowledge about the domain than is customarily given), and each seems to invest a class with additional power (as the identity assignment augments \( P(n,0) \)). It then becomes a matter of interest what other interpreted functions and predicates there might be which tell nothing more about an individual domain element, but which in some way disturb the power hierarchy.

A third interesting result about the class \( P(2b,0) \) is that it becomes universal if a chip is substituted for one of its bottom markers. The distinction is between requiring a marker to be in a particular relative position (namely, at the end of the data stored in a \( \text{pds} \)) and allowing it to move freely. This in some sense confirms the intuitive feeling that bound markers (like bottom markers) are less powerful than free markers which can be placed anywhere in the data.
(6.8) **Lemma:** \( P_{(2b,0)} \) becomes universal if one of its bottom markers is allowed to "float" (that is, if storage of elements beneath a bottom marker is permitted).

**Proof** The proof is contained in the Appendix.

6.3 **Other Possibly Significant Additions**

It can be readily deduced that program schema studies suggest all manner of interesting topics, some of which are under investigation by various researchers and some of which are as yet untouched. Two large and rather difficult areas are computational complexity as related to program schemata, and parallelism.

Before schemata can be of practical application to the optimization problem, as discussed in the introduction, it is imperative that a suitable cost function for judging among data structures and algorithms be developed. It is clear, for example, that a \( P_{(nb,0)} \) schema or a \( P_{Ae} \) schema can be precisely simulated by a schema in \( P_{(3b,0)} \), but what is not so evident is the magnitude of the penalty that must be paid for trading hardware resources (pds's, say) for increased time and storage requirements. Such analysis is quite difficult, but the payoff can be valuable; simulations using some form of Gödelization use exponentially increased space and
data structure access time over the subject schemata, and if such an increase were proved minimal, then a definite grounds for choosing one algorithm over another in a mechanical optimization could be provided.

Parallelism has been investigated rather carefully by Karp and Miller [Karp69], Ashcroft and Manna [Ash71], Slutz [Slutz70], and others, and quite difficult questions have been considered. Schematizing parallelism (abstracting the essence of the control pattern and of interprocess communication) still is not well understood, however, and the topic has not yet been attacked from a programming language viewpoint after the form of [Con72]. In this area schemata are valuable as a tool for understanding more about the mechanisms, pitfalls, and efficiencies of parallelism.

One of the aims for schema research is to draw the theoretical and the actual in computer science closer together, and to accomplish this end it is necessary to extend thinking in both directions. The similarity of the one two-way write-after-read tape configuration to a stack automaton and the resemblance between the class P and finite automata suggest that schema classes modeling other theoretical devices might prove instructive. A large omission with no analogue as yet is that of linearly bounded automata, whose work space is defined to be the number of cells occupied by the input. The extension to schemata comes most obviously in unbounded input
classes, where the amount of storage available could be restricted to \( k \cdot n \) (for input \( x_1, \ldots, x_n \)) or even \( f(n) \), for suitable \( f \) (corresponding to tape-bound computations of a Turing machine). The application of such \( k \)-limited or \( f \)-limited schemata is in modeling real programs whose workspace, say, is the area which the input occupies on a disk. Such an application has the flavor both of initialized input studies and of theoretical modeling. A further matter of interest in such a study lies in the uncovering of how an intermediate theoretical class translates into the schema power hierarchy.

With respect to the development of schemata toward increased verisimilitude, several conditions could be checked out and their effects noted. Among these are items such as multiple unbounded inputs, two-way tape-like inputs, types for variables, an interval timer (perhaps useful in modeling parallelism or partial basic functions and predicates), and associative memory. Additionally, the possible use of schemata as an interactive input vehicle for program source code should be investigated, especially because it is conjectured that with proper choice of schematizing primitives, such an approach could do much to encourage the type of structured programming approach recommended by Dijkstra [Dijk70].
6.4 **Overall Significance of This Work**

As can clearly be seen from the foregoing exposition, the intent of this study is not to wind up investigation of an area, but rather to investigate some items, such as queues, tapes, and unbounded input, in considerable depth and at the same time to indicate a variety of problems and research directions that remain open.

The introduction of the concept of an information flow approach to programming is important because of the potential understanding that a general theory of this type can yield. Information flow analysis may provide the basis for an entire programming method in which the necessary flow of data is specified (in an as yet undeveloped notation) and a compiler selects and sets up proper data structures. The application of information flow ideas to correctness proofs may also result in simplification of such techniques.

Significant points about information flow that have been demonstrated include the inherent power of persistence (non-destructive reading) when two persistent storage units are available, the distinction between potential flow (possible directions, etc.) and actual flow (which may involve additional control elements, encoding, and locally different ordering and flow), and the relationship of information flow to the control structure (the relationship is analogous to that between a canal system and its locks, sensors, signals,
and markers; such elements can be added or removed and cause some effect on the possible traffic, but the basic waterway configuration is not altered by such changes).

A new program data structure, the queue, has been introduced and investigated (illustrating how an information flow viewpoint can suggest a data structure and how such a structure can be evaluated in the light of schemata). The tape and the concept of unbounded input have been introduced as models close to reality in programming, and the effect of the upset of the generally strict finite-input hierarchy by the addition of unbounded input has been demonstrated and explored.

Finally, progress has been made in determining the "lay of the land" in schema power. Such study is essential for determining the extent and likely directions of expansion for the field (considerable suggestions have been offered to this end), and is important because of its implications for computer networks (the network controllers act as schemata and the terminals as basic functions and predicates) and eventual applications to mechanical optimization.

Specific important insights that have been achieved include the discovery that "reasonable" addition of infinite storage capacity (a single tape) to a finite storage schema class does not always result in increased power, the recognition of the noncongruence of the combination of schemata and
specific requirements for additive machine connection, and increased understanding of the importance of markers and their more restricted compatriots, chips and emptiness tests (also called bottom markers because of their functional equivalence).
BIBLIOGRAPHY


APPENDIX

The following appendix contains proofs of theorems and lemmas stated in Chapters 5 and 6. For a more detailed explanation of the background and significance of the theorems, the reader is encouraged to refer back to those chapters.

(5.1)  **Lemma:** If schema connections are defined as in Chapter 5 except that multiple messages are permitted on the communications queue, then the connection of two schemata in \( P \) (with markers) can simulate any schema in \( P \).

**Proof** A schema \( S' \) in \( P_{\Lambda \times \Lambda} \) is constructed to simulate a given schema \( S \) in \( P \). First, the statements of \( S \) are simplified somewhat as follows ( \( n \) new variables \( v_j' \) corresponding to the variables \( v_j \) of \( S \) are introduced):

Every statement \( \text{IF } V \equiv W \text{ THEN } S_1 \text{ ELSE } S_2 \) is replaced by

\[
\text{BEGIN } X + 0; \\
\text{LOOP: } \text{IF } W \equiv 0 \text{ THEN} \\
\text{IF } V \equiv 0 \text{ THEN GO TO EQUAL} \\
\text{ELSE GO TO UNEQUAL} \\
\text{ELSE IF } V \equiv 0 \text{ THEN GO TO UNEQUAL;} \\
X + X + 1; \quad \text{Equality or inequality is} \\
V + V \equiv 1; \quad \text{determined by successive sub-} \\
W + W \equiv 1; \quad \text{tractions, with the number of} \\
\text{GO TO LOOP; subtractions being saved in } X.
\]

160
EQUAL: IF X ≠ 0 THEN GO TO TRUESTMT;
    X + X := 1;
    V + V + 1;  The values of V and W are
    W + W + 1;  restored.
    GO TO EQUAL;

TRUESTMT: S1;
    GO TO OUT;

UNEQUAL: IF X ≠ 0 THEN GO TO FALSESTMT;
    X + X := 1;
    V + V + 1;  The values of V and W are
    W + W + 1;  restored.
    GO TO UNEQUAL;

FALSESTMT: S2;
    OUT:
    END

where X is a new integer variable not used in S and LOOP, EQUAL, TRUESTMT, UNEQUAL, FALSESTMT, and OUT are labels unique to each instance.

Every statement V + W + 1 is replaced by

BEGIN  V := 0;  V + V + 1;  X := 0;  V' := OMEGA;
    LOOP: IF W ≠ 0 THEN GO TO RESTORE;
    W + W := 1;  V is set equal to W + 1 by
    V + V + 1;  successive subtractions from
    X + X + 1;  W.  X is used to keep the
    GO TO LOOP;  value of W for later
    RESTORATION.

    RESTORE: IF X ≠ 0 THEN GO TO OUT;
    W + W + 1;
    X + X := 1;  The value of W is restored.
    GO TO RESTORE;
    OUT:
    END

Every statement V + W is replaced by similar code, except that V is initialized to 0 by deleting the initial statement V + V + 1.
Every statement \( V + W \hat{=} 1 \) is replaced by similar code except that \( V \) is initialized to 0 and the null statement labeled OUT is changed to be

IF \( V \equiv 0 \) THEN \( \Lambda \) ELSE \( V \hat{=} V \hat{=} 1 \).

Every statement \( V \hat{=} f(...) \) is replaced by

BEGIN
  LOOP: IF \( V \equiv 0 \) THEN GO TO OUT;
         \( V \hat{=} V \hat{=} 1; \) The integer value of \( V \) is
         GO TO LOOP; set to be 0.

  OUT: \( V' \hat{=} f(...) \)
END

Every statement \( V \hat{=} V + 1 \) is replaced by

BEGIN \( V \hat{=} V + 1; \) \( V' \hat{=} OMEGA \) END

All statements \( V \hat{=} 0 \) (including those inserted above) are changed to

BEGIN
  LOOP: IF \( V \equiv 0 \) THEN GO TO OUT;
         \( V \hat{=} V \hat{=} 1; \)
         GO TO LOOP;

  OUT:
END

All statements \( V \hat{=} V \hat{=} 1 \) not inserted by the above editing are replaced by the statement

IF \( V \equiv 0 \) THEN \( V' \hat{=} OMEGA \) ELSE \( V \hat{=} V \hat{=} 1 \)

No statement of the \( k \) statements making up the transformed \( S \) now uses any integer manipulation other than \( V \hat{=} V + 1 \), \( V \hat{=} V \hat{=} 1 \), and IF \( V \equiv 0 \) THEN \( S_1 \) ELSE \( S_2 \), and moreover, dot minus could be replaced by conventional minus.
In the communicating process formulation schemata $S_A$ and $S_B$ affect each other by means of their message queues, which are initially empty. Both $S_A$ and $S_B$ (which together form $S'$) are first set to be completely labeled copies of $S$ (modified as above). $S_A$ and $S_B$ take turns executing and keep the current Gödelization of integer variables in the message queue of one process or the other. The usual message (composed of multiple SENDs) appears as follows:

<table>
<thead>
<tr>
<th>M3</th>
<th>Gödelization</th>
<th>M2</th>
<th>unary indication of label of next statement to be executed</th>
<th>M1</th>
<th>current domain element contents of simple variables</th>
</tr>
</thead>
</table>

where $M1$, $M2$, and $M3$ are new distinguished markers.

New first statements of $S_A$

\[
\text{SEND(OMEGA, \ldots, OMEGA, M1, M, M2, OMEGA)}; \\
\text{\hspace{1cm}} \text{n times} \\
\text{\hspace{1cm}} \ell: \text{IF EMPTYMESSAGEQUEUE THEN GO TO } \ell \\
\text{\hspace{1cm}} \text{ELSE GO TO BRANCH;}
\]

and a new first statement of $S_B$

\[
\ell: \text{IF EMPTYMESSAGEQUEUE THEN GO TO } \ell \\
\text{ELSE GO TO BRANCH;}
\]

are inserted. In both $S_A$ and $S_B$

Every statement $\ell_i: V_j + V_j + 1$ is replaced by
BEGIN  SEND(V_1',...,V_n',M_1,M,...,M,M_2);
        \hspace{1cm} i+1 \text{ times}
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
RECEIVE(X);
UNTIL X = M_3 DO
BEGIN  SEND(OMEGA,...,OMEGA);
        \hspace{1cm} p_j \text{ times}
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
RECEIVE(X)
END;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
GO TO BRANCH
END

Every statement \( \ell_i : V_j \rightarrow V_j \pm 1 \) is replaced by

BEGIN  SEND(V_1',...,V_n',M_1,M,...,M,M_2);
        \hspace{1cm} i+1 \text{ times}
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
RECEIVE(X);
UNTIL X = M_3 DO
BEGIN  UNTIL NOT \hspace{1cm} \text{repeated}
        \hspace{1cm} \text{\textit{\(p_j\)-1 times}}
EMPTYMESSAGEQUEUE \hspace{1cm} \text{\textit{\(p_j\)-1 times}}
DO \Lambda;
RECEIVE(X);
:;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
RECEIVE(X)
END;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
GO TO BRANCH
END

Every statement \( \text{IF \( V_j \equiv 0 \) THEN } \ell_T : S_1 \text{ ELSE } \ell_F : S_2 \) is replaced by

BEGIN  SEND(V_1',...,V_n',M_1,M_2);
UNTIL NOT EMPTYMESSAGEQUEUE DO \Lambda;
RECEIVE(X);
LOOP: IF X = M3 THEN GO TO TRUE;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(X);
IF X = M3 THEN
    GO TO TRUE;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE
    DO Λ;
RECEIVE(X);
...
IF X = M3 THEN GO TO FALSE;
SEND(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(X);
GO TO LOOP;

TRUE: Vn+1 + M1; ... ; Vt + M1; Vt+1 + M2;
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
GO TO BRANCH;

FALSE: Vn+1 + M1; ... ; Vf + M1; Vf+1 + M2;
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
GO TO BRANCH;
END

In each schema S_A and S_B, an identical section labeled BRANCH is inserted:

BRANCH:
BEGIN UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(V_1');
...
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(V_n');
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(X);
UNTIL NOT EMPTYMESSAGEQUEUE DO Λ;
RECEIVE(Y);
IF Y = M2 THEN
    BEGIN SEND(V_1',...;V_n';M1,M,...,M,M2);
    UNTIL Y = M3 DO
BEGIN UNTIL NOT EMPTYPRIORITY do \lambda ;
RECEIVE(y);
SEND(y)
END;
UNTIL NOT EMPTYPRIORITY do \lambda ;
GO TO BRANCH
END;
UNTIL NOT EMPTYPRIORITY do \lambda ;
RECEIVE(y);
IF Y = M2 THEN GO TO \ell_1;
::
UNTIL NOT EMPTYPRIORITY do \lambda ;
RECEIVE(y);
IF Y = M2 THEN GO TO \ell_k;
IF \nu_{n+2} = M2 THEN GO TO \ell_1;
::
IF \nu_{n+k} = M2 THEN GO TO \ell_{k-1};
GO TO \ell_k
END

All integer arithmetic has thus been accurately mimicked and the result is established.

Q.E.D.

(6.1) **Theorem:** \( P \) without the identity function \( \equiv P \) with the identity function.

**Proof** From \( S \) in \( P \), \( S' \) in \( P \) without the identity function is created by replicating copies of \( S \) (which uses \( n \) simple variables) to keep track of all possible "same as" relations which are produced by assignments of the form \( \nu_i \leftrightarrow \nu_j \). A brute force method to accomplish the simulation and proof is to create \( n \) dummy variables \( D_1, \ldots, D_n \) in \( S' \) which are used to hold the actual values of the computation. Each of the \( n+n \) copies created
(each of the $n$ variables of $S$ can be the same as any one of the $n$ values contained in the $n$ dummy variables) is then indexed by $S[i_1,i_2,\ldots,i_n]$, where $i_j$ is the index of the dummy variable which is the same as variable $v_j$. Each statement of each copy is labeled $\ell[i_1,i_2,\ldots,i_n,t]$, where $t$ is the index of the statement in $S$. Then the following changes are made:

1) In copy $S[i_1,i_2,\ldots,i_n]$ every instance of variable $v_j$ is changed to $D_{ij}$, for all copies and all $j = 1,\ldots,n$.

2) Every statement $\ell[i_1,i_2,\ldots,i_n,t]$: $D_k + D_m$ is replaced by

   GO TO $\ell[i_1,\ldots,i_{k-1},i_m,i_{k+1},\ldots,i_n,t+1]$.

3) Every statement $\ell[i_1,i_2,\ldots,i_n,t]$: $D_{ik} + f(\ldots)$, where $f$ is a basic function, is changed as follows:

   a) if for each $j$, $j \neq k$ implies that $i_j \neq i_k$, then no change is necessary;

   b) otherwise, a $p$, $1 \leq p \leq n$, such that $i_j \neq p$ for any $j$ is chosen and the statement replaced by

   BEGIN $D_p + f(\ldots)$;
   GO TO $\ell[i_1,\ldots,i_{k-1},p,i_{k+1},\ldots,i_n,t+1]$
   END

The $k$ input variables $v_1, \ldots, v_k$, say, are renamed to be $D_1, \ldots, D_k$, and the initial state is then set to be
S[1,2,3,...,k,k+1,k+1,...,k+1].

Q.E.D.

One should notice that the above is not equivalent to simulating the domain element equality test proposed by Chandra and Manna [Chan71].

(6.2) **Theorem:** \( P_{\text{Ae}} \) **without the identity function** \( \equiv \) \( P_{\text{Ae}} \) **with the identity function.**

**Proof** For simplicity, a schema \( S \) in \( P_{\text{Ae}} \) is first transformed by a straightforward process so that all parameters to functions are simple variables and so that all array references are of the form \( A[v] \), where \( v \) is a simple variable. New variables may be introduced during such a transformation so that the resulting schema \( S' \) has \( n \) simple variables, say \( v_1, \ldots, v_n \).

\( S' \) is then altered so that the simple variables appear to be, respectively, elements \( A[1], \ldots, A[n] \) of the array \( A \) of \( S \) (for simplicity, \( S \) is assumed to have only one array; the simulation of many arrays using one actual array is discussed in Section 4 of [Con72]). Ordinary array references must then be "backed up" to higher positions in the array. Specifically, every reference to variable \( v_i \) is replaced by the array reference (with constant subscript) \( A[i] \), and every (previously existent) array reference \( A[j] \) is changed to \( A[j + n + 1] \). The resulting schema is designated \( S' \).
The simulation schema, built by making alterations to $S''$, employs arrays VALUES and SAMEAS, a simple variable TOP, and an initialized simple variable ZERO:

VALUES is used to store values calculated by the schema, as they occur.
SAMEAS[i] = k, where A[i] currently has (logically) the value stored in VALUES[k]. The entry in VALUES[0] is kept $\Omega$, and since $\Omega$ used as a subscript behaves as 0, uninitialized locations (for which SAMEAS[i] = $\Omega$) have the proper value of $\Omega$.

TOP is a counter which marks the last used element of VALUES. The first statement of $S''$ is TOP + ZERO + 1.

ZERO is a variable initialized to the subscript value 0.

The following transformations must be made in $S''$:

1) Statements incorporating a reference A[j + n + l] introduced above are changed to rephrase the subscript in terms of permitted operations. For example, if one such reference appears in a statement (the generalization to more than one reference is clear), the statement is replaced by the block
BEGIN  
  \[ j' = j + 1; \]
  \[ j' = j' + 1; \quad \text{repeated } n \text{ times} \]
  \[ \vdots \]
  \[ \ldots A[j'] \ldots \quad \text{statement originally using} \]
  \[ A[j + n + 1] \]
END

where \( j' \) is a new simple variable.

2) Every parameter \( A[i] \) to a basic function is replaced by \( \text{VALUES[SAMEAS[i]]} \).

3) Every statement \( A[v] = A[w] \) (by the earlier-described transformations \( w \) is never 0) is replaced by the block

BEGIN  
  \[ \text{SAMEAS[v]} = \text{ZERO} + 1; \]
  \[ \text{UNTIL \ sameas[v] = sameas[w] } \text{DO} \]
  \[ \text{SAMEAS[v]} = \text{SAMEAS[v]} + 1 \]
END

4) Every statement \( A[v] = f(\ldots) \) is replaced by the block

BEGIN  
  \[ \text{SAMEAS[v]} = \text{TOP} + 1; \]
  \[ \text{TOP} = \text{TOP} + 1; \]
  \[ \text{VALUES[TOP]} = f(\ldots) \]
END

All identity assignments are thus simulated and the theorem is established.

Q.E.D.

(6.3) Theorem: \[ P_{(n,0)} \quad \text{without the identity function} \]
\[ < \]
\[ P_{(n,0)} \quad \text{with the identity function.} \]

Proof The functional ModifiedleafTest can be
considered:

\[
\text{Modifiedleafest}(P,L,R,x,y,z) = \begin{cases} 
\text{if } P(y) \text{ and not } P(z) \text{ then} \\
\text{Leafest}(P,L,R,x) \\
\text{else } \Omega
\end{cases}
\]

Theorem 3.5 of [Bro72] shows that this functional can be computed in $P(n,0)$ . Now if it is supposed, as usual, that $P(\Omega) = \text{false}$ and that $P(y)$ and $\text{not } P(z)$ , then, essentially, Leafest must be computable in $P(n,0)_C$ (the class $P(n,0)$ with one chip) if it is computable without the identity function. This follows because without the identity assignment the single input domain element $y$ cannot be duplicated. But then by the helping lemma below and the same arguments used in Lemma 3.3 of [Bro72], Leafest would be computable in $P$ , which contradicts known results ([Pat70]).

Q.E.D.

(6.3a) **Helping Lemma:** There is a locator in $P$ for any schema $S$ in the class $P(n,0)_C$ .

**Proof** If $S$ is a schema in $P(n,0)_C$ with $|S|$ statements, autonomous behavior is considered. As long as the chip is not placed in a pds, the schema behaves as a $P(n,0)$ schema (the location of the chip can be kept by replications of $S$ ) and since there are at most (for a $v$-variable schema) $T = |S| \cdot 2^v$ internal configurations, if $S$ executes more than $T$ statements while the chip is in
simple variables, it is looping.

If the chip is pushed onto a pds j and stays there for more than |S| statement executions, the schema is again in a loop from which it will not exit. The maximum detectable depth of a pds is thus less than |S|.

Consequently, the number of detectable internal states of the schema is less than \( U = |S| + T \). Under constant predicates \( S \) executes \( W \) (say) statements, \( W < U \), and then either halts or enters an infinite loop. In either case, as shown in the proof of Lemma 3.2 of [Bro72], a locator in \( P \) exists for \( S \) since a fixed number of locations is sufficient to simulate the n pds's.

Q.E.D.

(6.4) Lemma: \( P(2b,0) + (1,0) \) is universal.

Proof The result is a consequence of the conclusions in Section 10 of [Con72] that \( P(1,0) + \biguplus \) is a universal class, and of the ability of \( P(2b,0) \) to perform integer arithmetic. The extra pds PD3 is used as the pds of \( P(1,0) + \biguplus \), and the fixed number of integer variables of \( S \) in \( P(1,0) + \biguplus \) are stored using Gödelization as a length of cells in one of the bottom-testable pds's PD1 and PD2 of \( S' \) in \( P(2b,0) + (1,0) \). Thus \( L \), the number of cells stored (canonically) in PD1, is \( p_1 v_1 \cdot \ldots \cdot p_n v_n \), where \( v_1, \ldots, v_n \) are the integer variables of \( S \). Incrementing \( v_i \) is accomplished by pouring PD1 to PD2, adding \( p_i - 1 \)
cells for every cell transferred, and then pouring PD2 back to PD1. Testing $v_1$ for zero is accomplished by the usual modulus method in which PD1 is poured into PD2 as $z = |PD1| \mod p_i$ is computed. PD2 is then poured back to PD1 and the zero test is considered true iff $z \neq 0$.

Decrementing $v_1$ is achieved by testing $v_1$ for zero as above; if $v_1 \neq 0$ then $|PD1|$ is divided by $p_i$ by pouring PD1 to PD2, but transferring only one out of every $p_i$ cells that are removed from PD1.

Q.E.D.

(6.6) **Lemma:** $P(2b,0)$ with the predicate TALLER is universal.

**Proof** As mentioned in the text, TALLER(PD1,PD2) is true iff the height of pads PD1 is greater than that of pads PD2. Equality of pads heights can be tested by a simple logical conjunction, and by this means the height of pads PD1 can be duplicated as a number of cells in PD2 without disturbing PD1. The proof of universality then centers around how to use the predicate to permit simulation of three bottom-testable pads's, using interleaved Gödelization to store the elements (canonically kept in PD1). Testing simulated pads $j$ for emptiness is carried out by the usual modulus computation. Pushing pads $j$ is achieved by the usual method of multiplying the length of PD1 and percolating elements. The multiplication is accomplished by
UNTIL EMPTYPDS(PD1) DO
BEGIN  POP(PD1,X);
       PUSH(PD2,X)
END

UNTIL NOT TALLER(PD1,PD2) AND
     NOT TALLER(PD2,PD1) DO
     PUSH(PD1,OMEGA);

UNTIL EMPTYPDS(PD1) DO
BEGIN  POP(PD1,X);
       PUSH(PD2,X);
       repeated
          P_j - 1
         times
END

UNTIL EMPTYPDS(PD2) DO
BEGIN  POP(PD2,X);
       PUSH(PD1,X)
END

The Gödelization is inverted.

The length of the Gödelization is replicated.

The new padding cells are attached.

The Gödelization is restored to its canonical storage.

A similar method is used for popping, except that after inversion and length replication, PD1 is popped until it is empty, and after each pop $p_j - 1$ elements are removed from the top of PD2.

Q.E.D.

(6.8) Lemma: $P(2b,0)$ becomes universal if one of its
       bottom markers is allowed to "float" (that is,
       if storage of elements beneath a bottom marker
       is permitted).

Proof  The result again follows from the facts that
       $P(1,0) + \mathbb{N}$ is universal and that $P(2b,0)$ can perform
       integer arithmetic. The proof of Lemma 6.4 can be restated
       so that the chip plays the role of the bottom marker in PD2
       and PD1 canonically contains the Gödelization of the
       integer variables. Since the manipulation of an integer
variable and the need to reference the pds PD in a $P(1,0) + \mathbb{N}$ schema are always disjoint in time, the part of PD2 below the chip can be used to keep PD (when PD is referenced the integer variable Gödelization is transferred to PD1 and the chip is temporarily removed).

Gödelization is kept canonically here.

\[ \text{PD1} \quad \text{PD2} \]

This space is used as necessary during integer manipulations.

PD is stored here.

Q.E.D.