LOGSPACE and PTIME Characterized by Programming Languages

Neil D. Jones [1]

DIKU, University of Copenhagen, Denmark
E-mail: neil@diku.dk

A programming approach to computability and complexity theory yields more natural definitions and proofs of central results than the classical approach. Further, some new results can be obtained using this viewpoint. This paper contains new *intrinsic* characterizations of the well-known complexity classes PTIME and LOGSPACE, with no externally imposed resource bounds on time or space. LOGSPACE is proven identical with the decision problems solvable by *read-only* imperative programs on Lisp-like lists; and PTIME is proven identical with the problems solvable by *recursive* read-only programs.

Key words: Complexity, read-only or cons-free programs, PTIME, LOGSPACE.

1 Introduction

Thesis. We maintain that Computability and Complexity theory, and Programming Language and Semantics (henceforth CC and PL) have much to offer each other, in both directions [2]. CC has a breadth, depth, and generality not often seen in PL, and a tradition for posing (and occasionally answering) sharply defined *open problems* of community-wide interest. PL has a firm grasp of algorithm design, presentation, and implementation, and several well-developed frameworks for making precise semantic concepts over a wide range of PL concepts (functional, imperative, control flow operators, communication/concurrency, object-orientation, and much more).

[1] This research was partially supported by the Danish Research Council (*DART* project), and the Esprit BRAs *Semantique* and *Atlantique*. An earlier version of this paper appeared in *Electronic Notes in Theoretical Computer Science*, proceedings of the 1995 meeting *Mathematical Foundations of Programmming Semantics*.

[2] This theme is further developed in [12], which is an introduction to computability and complexity using programming-related models.

Some concrete connections. It is natural in PL to have efficient built-in *data construction and decomposition* operators: these are just examples of the "pairing functions" known in CC from the 1930s. We take them as primitives, as in the Lisp language. The main results of this paper are simple "intrinsic" characterizations of the well-known problem classes LOGSPACE and PTIME, in programming terms and without external imposition of space or time bounds; and new insights into the role of persistent (as opposed to evanescent) storage. One effect is that the use of PL concepts lead (at least for Computer Scientists) to more understandable statements of theorems and proofs in CC, and to stronger results.

Some interesting questions. Further, a number of old CC questions take on new life, and natural new questions arise. An important question category is: what is the effect of *the programming styles we employ* (functional, imperative, etc.) on the *efficiency of the programs we can possibly write?*

A puzzling tradeoff: we will see that a problem is solvable in polynomial time just in case it is solvable by recursive read-only program. Paradoxically, recursive read-only programs often run in exponential time (not a contradiction, since they can be simulated in polynomial time by memoization.) This tradeoff indicates a tension between running time and memory space which seems worth further investigation.

2 Programming languages and complexity classes

2.1 Programming languages

Definition 2.1 A *programming language* L consists of two sets, L-*programs* and L-*data*, together with L's *semantic function*, which associates with every $\mathtt{p} \in$ L-*programs* a corresponding (partial) input-output function

$$[\![\mathtt{p}]\!]^{\mathtt{L}}(\_) : \mathtt{L}\text{-}data \to \mathtt{L}\text{-}data_{\perp}$$

We are concerned with time and space bounded computations. A definition encompassing both follows:

Definition 2.2 A *resource-usage measure* on L-programs is a partial function

$$usage^{\mathtt{L}}_{\mathtt{p}}(\_) : \mathtt{L}\text{-}data \to \{\perp, 0, 1, 2, \ldots\}$$

This paper treats only decision problems, hence the following. Note that program $\mathtt{p}$ must terminate on all inputs.

Definition 2.3 Let $\mathtt{true}, \mathtt{false} \in \mathtt{L}$-*data* be two distinct data values. An $\mathtt{L}$-program $\mathtt{p}$ *decides* a subset A of $\mathtt{L}$-*data* if for any $\mathtt{d} \in \mathtt{L}$-*data*

$$[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = \begin{cases} \mathtt{true} & \text{if} \quad \mathtt{d} \in A \\ \mathtt{false} & \text{if} \quad \mathtt{d} \in \mathtt{L}\text{-}data \backslash A \end{cases}$$

□

The requirements above are satisfied by any reasonable programming language. Additional naturality restrictions are often imposed, for example the following:

(i) *Turing completeness*: a partial function $f : \mathtt{L}$-$data \to \mathtt{L}$-$data_\perp$ is computable if and only if $f = [\![\mathtt{p}]\!]^{\mathtt{L}}$ for some $\mathtt{L}$-program $\mathtt{p}$.

(ii) *Universal function property*: $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d})$, regarded as a two-argument partial function of $\mathtt{p}$ and $\mathtt{d}$, is a computable partial function.

(iii) Suppose $\mathtt{L}$-programs may have multiple inputs. The *s-m-n property*, or program specialization: for any $m, n \geq 0$ there exists a computable total function

$$s_m^n : \mathtt{L}\text{-}programs \times \mathtt{L}\text{-}data^m \to \mathtt{L}\text{-}programs$$

such that for any m+n-input $\mathtt{L}$-program $\mathtt{p}$ and inputs $\mathtt{s}_1, \ldots, \mathtt{s}_m, \mathtt{d}_1, \ldots, \mathtt{d}_n \in \mathtt{L}$-*data*, it holds that

$$[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{s}_1, \ldots, \mathtt{s}_m, \mathtt{d}_1, \ldots, \mathtt{d}_n) = [\![s_m^n(\mathtt{p}, \mathtt{s}_1, \ldots, \mathtt{s}_m)]\!]^{\mathtt{L}}(\mathtt{d}_1, \ldots, \mathtt{d}_n)$$

(iv) For any $\mathtt{p} \in \mathtt{L}$-*programs* and $\mathtt{d} \in \mathtt{L}$-*data*, $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = \perp$ if and only if $usage_{\mathtt{p}}^{\mathtt{L}}(\mathtt{d}) = \perp$.

(v) This set is decidable for any $\mathtt{p} \in \mathtt{L}$-*data*:

$$\{(\mathtt{p}, \mathtt{d}, n) \mid usage_{\mathtt{p}}^{\mathtt{L}}(\mathtt{d}) \leq n\}$$

Properties (i)–(iii) state that language $\mathtt{L}$ is an *acceptable enumeration* of the partial recursive functions [15], and properties (iv), (v) state that *usage* is a complexity measure acceptable in Blum's sense [2]. These properties are easily seen to hold for the languages we will introduce; proofs are natural but omitted for brevity.

Definition 2.4 Let languages $\mathtt{L}$ and $\mathtt{M}$ have the same data. Then $\mathtt{L}$ *can simulate* $\mathtt{M}$, written $\mathtt{M} \preceq \mathtt{L}$, if for every $\mathtt{M}$-program $\mathtt{p}$ there exists an $\mathtt{L}$-program $\mathtt{q}$ such that $[\![\mathtt{p}]\!]^{\mathtt{M}} = [\![\mathtt{q}]\!]^{\mathtt{L}}$. Language $\mathtt{L}$ *is equivalent to* $\mathtt{M}$, written $\mathtt{L} \equiv \mathtt{M}$, if $\mathtt{L} \preceq \mathtt{M}$ and $\mathtt{M} \preceq \mathtt{L}$. □

By Property i, any two languages satisfying the conditions above can simulate one another. Our concern, however, will be efficient mutual simulations. Given a bound on resources, e.g., n^3, complexity is concerned with the set of all problems solvable by programs `p` that run within that bound, when applied to any possible input.

We diverge slightly from custom, and define a complexity class to be a set of *programs* that run within a certain resource bound, rather than a set of *problems* that can be solved within that bound [3]. Thus we can and will regard an `L` complexity class as a *sublanguage* of `L`, restricted to all programs satisfying the given resource bounds, with programs having exactly the same semantics as they would in `L`.

It is customary to measure resource usage as a function of the input *size* $|\mathtt{d}|$ (where $|\mathtt{d}|$ is length, number of symbols, etc. of input `d`), rather than the input itself. (Concrete size measures will be seen shortly.)

Definition 2.5 Given programming language `L` with resource-usage measure *usage*, and a function $f : I\!N \to I\!N$, define

$$\mathtt{L}^{usage(f)} = \{\mathtt{p} \in \mathtt{L}\text{-}program \mid usage^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) \leq f(|\mathtt{d}|) \text{ for all } \mathtt{d} \in \mathtt{L}\text{-}data\}$$

This paper concerns two resource-usage measures: *time* and *space* (concrete definitions soon to come).The problems decidable in, respectively, polynomial time and logarithmic space, will be central:

$$\mathtt{L}^{ptime} = \bigcup_{a,b=0}^{\infty} \mathtt{L}^{time(\lambda n\,.\,a+n^b)}$$

$$\mathtt{L}^{log\,space} = \bigcup_{k=1}^{\infty} \mathtt{L}^{space(\lambda n\,.\,k \log n)}$$

□

3 The `WHILE` language and Turing machines

3.1 The `WHILE` *language*

We introduce a simple programming language called `WHILE`, in essence a small subset of Pascal or Lisp. Why just this language? Because `WHILE` seems to

[3] The difference is inessential, e.g. the well-known class PTIME is exactly the set of problems solvable by programs in $\mathtt{WHILE}^{ptime}$ as defined below.

have just the right mix of expressive power and simplicity for our purposes. *Expressive power* is important for carrying out constructions, e.g., or showing how one program can simulate another. *Simplicity* is also essential to prove theorems about programs and their behavior. This argues against larger, more powerful languages, since proofs about them would simply be too complex to be easily understood.

3.1.1 Syntax of WHILE *data and programs*

Definition 3.1 Let $A = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ be some finite set. Then

(i) $\mathbb{D}$ is the smallest set satisfying $\mathbb{D} = (\mathbb{D} \times \mathbb{D}) \cup A$. The *pairing* operation yields value $(\mathtt{d}_1.\mathtt{d}_2)$ when applied to values $\mathtt{d}_1$, $\mathtt{d}_2$.
(ii) The *size* $|d|$ of a value $\mathtt{d} \in \mathbb{D}$ is defined as follows: $|\mathtt{d}| = 1$ if $\mathtt{a} \in A$, and $1 + |\mathtt{d}_1| + |\mathtt{d}_2|$ if $\mathtt{d} = (\mathtt{d}_1.\mathtt{d}_2)$.

□

Values in the set $\mathbb{D}$ are built up from *atoms* in A by finitely many applications of the *pairing* operation "cons". A value $\mathtt{d} \in \mathbb{D}$ is thus a binary tree with atoms as leaf labels. An example, written in "fully parenthesized form": `((a.((b.nil).c)).nil)`. In formal constructions we will only use a single atom, called `nil` (so in fact $A = \{\mathtt{nil}\}$), but will for readability's sake use more atoms in examples. There is no loss of generality, as multiple-atom structures can be encoded into ones using only `nil` with no loss of information, the overhead being a multiplication of run time by a constant.

A compact linear notation for values: Unfortunately it is hard to read deeply parenthesized structures (one has to resort to counting), so we will use a more compact "list notation" taken from the Lisp and Scheme languages, in which

`()`	stands for	`nil`
$(\mathtt{d}_1 \cdots \mathtt{d}_n)$	stands for	$(\mathtt{d}_1.(\mathtt{d}_2.\cdots(\mathtt{d}_n.\mathtt{nil})\cdots))$

The syntax of `WHILE` programs is given by the "informal syntax" part of Figure 3.1. Programs manipulate tree structures built by `cons` from atoms. Operations `hd` and `tl` (head, tail) decompose such structures. In tests, `nil` serves as "false," and anything else serves as "true." For readability we will often write `false` in place of `nil`, and `true` in place of `(nil.nil)`. For an example, the following program, `reverse`:

P : Program	::=	`read X; C; write Y`
C : Command	::=	`Z := E`
	|	`C1; C2`
	|	`if E then C1 else C2`
	|	`while E do C`
E : Expression	::=	`Z` (any variable)
	|	`D`
	|	`cons E1 E2`
	|	`hd E`
	|	`tl E`
X, Y, Z : Variable	::=	`X0` | `X1` | ...
D : Data-value	::=	`nil` | `(D.D)`

Figure 3.1: WHILE *program syntax*

```
read X;
  Y := nil;
  while X do { Y := cons (hd X) Y;  X := tl X };
write Y
```

satisfies $[\![\texttt{reverse}]\!](\texttt{a.(b.(c.nil))}) = \texttt{(c.(b.(a.nil)))}$ or, in the compact notation, $[\![\texttt{reverse}]\!](\texttt{a b c}) = \texttt{(c b a)}$. In both cases the outermost parenthesis pair has been omitted.

3.1.2 Semantics of WHILE *programs*

Informally, the net effect of *running a program* p is to compute a partial function $[\![\texttt{p}]\!]^{\texttt{WHILE}} : \mathbb{D} \to \mathbb{D}_\perp$, where $\mathbb{D}_\perp$ abbreviates $\mathbb{D} \cup \{\perp\}$. Control structures are *sequential composition* C1;C2, the *conditional* `if E then C1 else C2`, and the *while loop* `while E do C`. In tests, `nil` serves as "false," and anything else serves as "true." A formal definition of the semantics:

Definition 3.2 Consider a WHILE program p of form `read X; C; write Y`. Let $\text{Vars}(\texttt{p}) = \{\texttt{X}, \ldots, \texttt{Y}\}$ be the set of all variables in p.

(i) A *store* σ for p is by definition a function from Vars(p) to elements of $\mathbb{D}$. More generally, $\text{Store}^{\texttt{p}} = \text{Vars}(\texttt{p}) \to \mathbb{D}$ is the set of all stores for p.

(ii) The *initial store* $\sigma_0^{\texttt{p}}(\texttt{d})$ for inputs $\texttt{d} \in \mathbb{D}$ binds X to d and all else to `nil`:

$$\sigma_0^{\texttt{p}}(\texttt{d}) = [\texttt{X} \mapsto \texttt{d}, \texttt{Z} \mapsto \texttt{nil}, \ldots, \texttt{Y} \mapsto \texttt{nil}]$$

Figure 3.2 contains the three semantic functions $\mathcal{E}, \mathcal{C}$, and $\mathcal{P}$ ($\mathcal{C}$ and $\mathcal{P}$ are partial) for WHILE programs, with types

$$
\begin{array}{lll}
\mathcal{E}[\![\texttt{X}]\!]\sigma & = \sigma(\texttt{X}) & \\
\mathcal{E}[\![\texttt{d}]\!]\sigma & = \texttt{d} & \text{if } \texttt{d} \in \mathbb{D} \\
\mathcal{E}[\![\texttt{cons E1 E2}]\!]\sigma & = (\texttt{d}_1.\texttt{d}_2) & \text{if } \mathcal{E}[\![\texttt{E1}]\!]\sigma = \texttt{d}_1 \text{ and } \mathcal{E}[\![\texttt{E2}]\!]\sigma = \texttt{d}_2 \\
\mathcal{E}[\![\texttt{hd E}]\!]\sigma & = \texttt{d}_1 & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = (\texttt{d}_1.\texttt{d}_2) \\
\mathcal{E}[\![\texttt{hd E}]\!]\sigma & = \texttt{nil} & \text{otherwise} \\
\mathcal{E}[\![\texttt{tl E}]\!]\sigma & = \texttt{d}_2 & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = (\texttt{d}_1.\texttt{d}_2) \\
\mathcal{E}[\![\texttt{tl E}]\!]\sigma & = \texttt{nil} & \text{otherwise} \\
\mathcal{E}[\![\texttt{X := E}]\!]\sigma & = \sigma[\texttt{X} \mapsto \texttt{d}] & \text{if } \mathcal{E}[\![\texttt{E}]\!] = \texttt{d} \\
 & & \\
\mathcal{C}[\![\texttt{C1 ; C2}]\!]\sigma & = \sigma' & \text{if } \mathcal{C}[\![\texttt{C1}]\!]\sigma = \sigma'' \text{ and } \mathcal{C}[\![\texttt{C2}]\!]\sigma'' = \sigma' \\
\mathcal{C}[\![\texttt{if E then C1 else C2}]\!]\sigma & = \sigma' & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma \neq \texttt{nil} \text{ and } \mathcal{C}[\![\texttt{C1}]\!]\sigma = \sigma' \\
\mathcal{C}[\![\texttt{if E then C1 else C2}]\!]\sigma & = \sigma' & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{nil} \text{ and } \mathcal{C}[\![\texttt{C2}]\!]\sigma = \sigma' \\
\mathcal{C}[\![\texttt{while E do C}]\!]\sigma & = \sigma & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = \texttt{nil} \\
\mathcal{C}[\![\texttt{while E do C}]\!]\sigma & = \sigma' & \text{if } \mathcal{E}[\![\texttt{E}]\!]\sigma = (\texttt{d}_1.\texttt{d}_2) \\
 & & \quad \text{and } \mathcal{C}[\![\texttt{C}]\!]\sigma = \sigma'' \\
 & & \quad \text{and } \mathcal{C}[\![\texttt{while E do C}]\!]\sigma'' = \sigma'
\end{array}
$$

$$\mathcal{P}[\![\texttt{read X; C; write Y}]\!](\texttt{d}) = \texttt{d}' \text{ if } \mathcal{C}[\![\texttt{C}]\!](\sigma_0^{\texttt{P}}(\texttt{d})) = \sigma(\texttt{Y})$$

Figure 3.2: Semantics of WHILE *programs.*

$$
\begin{array}{lll}
\mathcal{E} : \texttt{Expression} & \to & (\text{Store}^{\texttt{P}} \to \mathbb{D}) \\
\mathcal{C} : \texttt{Command} & \to & (\text{Store}^{\texttt{P}} \to \text{Store}^{\texttt{P}}_{\perp}) \\
\mathcal{P} : \texttt{Program} & \to & (\mathbb{D}^m \to \mathbb{D}_{\perp})
\end{array}
$$

Function $\mathcal{E}$ evaluates expressions. Given a store σ containing the values of the variables in an expression E, $\mathcal{E}$ maps E and σ into the value $\mathcal{E}[\![\texttt{C}]\!]\sigma$ in $\mathbb{D}$ that E denotes. Suppose command C has a number of assignments that alter the store. Given a store σ, function $\mathcal{C}$ maps the command and the current store into a new store $\mathcal{C}[\![\texttt{C}]\!]\sigma = \sigma' \in \text{Store}^{\texttt{P}}$. If command C does not terminate on the given store σ, then $\mathcal{C}[\![\texttt{C}]\!]\sigma$ is undefined, written: $\mathcal{C}[\![\texttt{C}]\!]\sigma = \perp$. Finally, $\mathcal{P}$ maps a program and given value d for the input variables into a value $\mathcal{P}[\![\texttt{p}]\!](\texttt{d}) = \mathcal{C}[\![\texttt{C}]\!](\sigma_0^{\texttt{P}}(\texttt{d}))$ in $\mathbb{D}$ if the program terminates, else $\perp$. The meaning of a program is written as $\mathcal{P}[\![\texttt{p}]\!] : \mathbb{D} \to \mathbb{D}_{\perp}$, and $\mathcal{P}[\![\texttt{p}]\!]$ will also be written as $[\![\texttt{p}]\!]^{\texttt{WHILE}}$, or even $[\![\texttt{p}]\!]$ if the language is clear from context. □

For brevity, running time is only informally defined. A later section on implementation justifies the reasonability of this definition—natural with the data-sharing implementation techniques used in Lisp and other functional languages.

Definition 3.3 The running time $time_{\texttt{p}}(\texttt{d}) \in \{\perp, 0, 1, 2, \ldots\}$ is obtained by counting 1 every time any of the following is performed while computing $[\![\texttt{p}]\!](\texttt{d})$

as defined in the semantics: a variable or constant reference; an operation `hd`, `tl`, `cons`, or `:=` is applied; or a test in an `if` or `while` command. Its value is $\perp$ if the computation does not terminate. □

3.1.3 The GOTO *variant of* WHILE

The `WHILE` language has both "structured" syntax and data. This is convenient for programming, but when constructing one program from another it will often be convenient to use a lower-level "flow chart" syntax in which a program is a sequence `p = 1:I1 2:I2 ... m:Im` of labeled instructions, executed serially except as redirected by control transfers. Instructions are of the form `X:=nil`, `X:=Y`, `X:=cons Y Z`, `X:=hd Y`, `X:=tl Y`, or `if X goto` ℓ `else` ℓ'. (Note that the syntax of `GOTO` expressions is significantly more limited than that of `WHILE` expressions.)

The semantics is natural and so not presented here. Following is the `GOTO` equivalent of the `reverse` program above. [4]

```
0: read X;
1: if X goto 2 else 6
2: Z := hd X;
3: Y := cons Z Y;
4: X := tl X;
5: goto 1;
6: write Y
```

It is easy to see that any `WHILE` program can be translated into an equivalent `GOTO` program running at most a constant factor slower (measuring `GOTO` times by the number of instructions executed). Conversely, any `GOTO` program can be translated into an equivalent `WHILE` program running at most a constant factor slower (the factor may depend on the size of the `GOTO` program).

3.2 Off-line Turing machines

A `TM`-program is a traditional off-line Turing machine with a two-way read-only *input tape*, and a two-way read-write *work tape*. By definition `TM-`*data* $= \{\mathtt{0},\mathtt{1}\}^*$. Since our aims concern only *decision powers* and not computation of functions, a Turing machine output will be a single bit `0` or `1`. Extension to outputs in $\{\mathtt{0},\mathtt{1}\}^*$ is routine, by a one-way write-only output tape.

A `TM`-program is a sequence `p = 1:I1 2:I2 ... m:Im`. The instructions `I`ℓ

[4] Where `goto 1` abbreviates `if X goto 1 else 1`.

are as follows, where subscript 1 indicates that the input tape 1 is involved; or 2 indicates that work tape 2 is involved. Instruction syntax:

Tape 1:	I	::=	$\texttt{right}_1 \mid \texttt{left}_1 \mid \texttt{if}_1\ \texttt{S goto}\ \ell\ \texttt{else}\ \ell'$
Tape 2:	I	::=	$\texttt{right}_2 \mid \texttt{left}_2 \mid \texttt{if}_2\ \texttt{S goto}\ \ell\ \texttt{else}\ \ell' \mid \texttt{write}_2\ \texttt{S}$
Symbols:	S	::=	0 | 1 | B
Strings:	L,R	::=	$\varepsilon \mid \texttt{L S}$

A tape together with its scanning position will be written as $\texttt{L}_1\underline{\texttt{S}}_1\texttt{R}_1$, where the underline indicates the scanned position. As usual the tape is extensible — a new blank appears when a move is made beyond the end of the tape. A *total state* is a triple $s = (\ell, \texttt{L}_1\underline{\texttt{S}}_1\texttt{R}_1, \texttt{L}_2\underline{\texttt{S}}_2\texttt{R}_2)$ whose first component ℓ is the number of the instruction about to be executed, and whose second and third components describe both of the tapes, and underlines mark their scanning positions.

The semantics of the individual instructions is a state transition relation $s \to s'$ (actually a function), as is usual for Turing machines. For example, instruction $\texttt{1: right}_2$ causes transition from state $(\texttt{1}, \texttt{B}\underline{\texttt{1}}\texttt{0}, \texttt{B0}\underline{\texttt{1}}\texttt{11B})$ to $(\texttt{2}, \texttt{B}\underline{\texttt{1}}\texttt{0}, \texttt{B01}\underline{\texttt{1}}\texttt{1B})$, or from $(\texttt{1}, \texttt{B}\underline{\texttt{1}}\texttt{0}, \texttt{B0}\underline{\texttt{1}})$ to $(\texttt{2}, \texttt{B}\underline{\texttt{1}}\texttt{0}, \texttt{B01}\underline{\texttt{B}})$. We assume the program never attempts to move right or left beyond the blanks that delimit the input, unless a nonblank symbol has first been written. [5]

A *computation* on input $\texttt{d} \in \{\texttt{0},\texttt{1}\}^*$ is a sequence $s_0 \to s_1 \to \ldots \to s_n$ where $s_i \to s_{i+1}$ for $0 \le i < n$, and s_0 equals $(\texttt{1}, \underline{\texttt{B}}\texttt{d}, \underline{\texttt{B}})$, and s_n has instruction $\texttt{m+1}$ as first component (the "program end"). The output for input d is defined by:

$$[\![\texttt{p}]\!](\texttt{d}) = \begin{cases} \bot & \text{if p has no computation on input d, else} \\ \texttt{1} & \text{if p's final state for input d scans worktape symbol 1} \\ \texttt{0} & \text{otherwise} \end{cases}$$

We define the *space usage* of a total state $s = (\ell, \texttt{L}_1\underline{\texttt{S}}_1\texttt{R}_1, \texttt{L}_2\underline{\texttt{S}}_2\texttt{R}_2)$ by $|s| = |\texttt{L}_2\texttt{S}_2\texttt{R}_2|$, formally expressing that only the symbols on "work" tape 2 are counted, and not those on tape 1. Finally, we define

$$\mathit{space}^{\texttt{TM}}_{\texttt{p}}(\texttt{d}) = \begin{cases} \max\{|s_i|\} & \text{if } s_0 \to \ldots \to s_n \text{ is p's computation on d} \\ \bot & \text{if p does not terminate on input d} \end{cases}$$

[5] This condition simplifies constructions, and causes no loss of generality in computational power, or increase in time beyond a constant factor.

4 Read-only programs decide exactly LOGSPACE

The class LOGSPACE plays a central role in complexity theory, as it is the beginning of the much-studied hierarchy LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ PTIME $\subseteq$ NPTIME $\subseteq$ PSPACE $\subseteq \ldots$ Further, all known reductions used to show familiar combinatorial problems complete for these classes can be carried out by logspace-bounded Turing machines.

LOGSPACE is in a sense the smallest natural complexity class, because a machine needs at least logarithmic storage in order to "remember" a position in its input string or to count a number of input symbols (for instance, to decide whether its input has the form $\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n$).

Despite the naturality and centrality of the class LOGSPACE, its Turing machine definition seems artificial due to hardware restrictions on the number of tapes, their usage, and the external restriction on the run-time length of the work tape. A somewhat more natural definition comes from a "folklore theorem:" that logspace Turing machines have the same decision power as two-way multihead finite automata (read-only) — machines that can "see but not touch" their input.

In this section we give a simpler and still more natural "look but not touch" characterization: LOGSPACE is exactly the set of problems decidable by read-only WHILE programs. More formally, let $\mathtt{WHILE}^{\setminus\mathtt{cons}}$ be the language identical to WHILE but restricted to programs not containing cons. We will prove the following:

Theorem 4.1 $\mathtt{WHILE}^{\setminus\mathtt{cons}} \equiv \mathtt{TM}^{logspace}$ □

4.1 About the simulations

A bijection between $\{\mathtt{0},\mathtt{1}\}^*$ and a subset of $\mathbb{D}$. As defined, Turing machine and WHILE inputs are not the same. We resolve this by restricting $\mathbb{D}$ to a subset $\mathbb{D}_{01}$ isomorphic with $\{\mathtt{0},\mathtt{1}\}^*$. Define the coding $c : \{\mathtt{0},\mathtt{1}\}^* \to \mathbb{D}$ by $c(\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n) = (\mathtt{a}_1'\ \mathtt{a}_2'\ \ldots\ \mathtt{a}_n')$ where $\mathtt{0}' = \mathtt{nil}$ and $\mathtt{1}' = (\mathtt{nil}.\mathtt{nil})$. Define $\mathbb{D}_{01}$ to be the range of c. An example using Lisp list notation:

$$c(\mathtt{001}) = (\mathtt{0}'\mathtt{0}'\mathtt{1}') = (\mathtt{nil\ nil\ (nil.nil)})$$

Relating WHILE and Turing machine states. Given a WHILE program, a Turing machine simulating it will store on its work tape the pointer values of all the WHILE variables. Conversely, given a Turing program, a WHILE program

simulating it will encode, by means of pointer values, the current contents of the Turing machine's work tape. We first analyze their numbers of states.

Turing machine: Let p have m instructions, work tape storage bound $k \log n$, and input $\mathtt{a}_1\mathtt{a}_1 \ldots \mathtt{a}_n \in \{\mathtt{0},\mathtt{1}\}^*$. Then p can enter at most

$$m(n+2)3^{k \log n} = m(n+2)n^{k \log 3}$$

different total states. For fixed p this number is $O(n^{1+k \log 3})$.

Read-only WHILE program: Let q have m' instructions, k' variables, and input list $(\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n) \in \mathbb{D}_{\mathtt{01}}$. During the computation every q-variable X value must be a pointer to one of:

(i) The root of a suffix $(\mathtt{a}_i\mathtt{a}_{i+1} \ldots \mathtt{a}_n)$ at a position i along the "spine" of the input list; or
(ii) The root of (nil.nil), the coding $c(\mathtt{1})$ of some $a_i = \mathtt{1}$; or
(iii) The atom nil. [6]

Thus each variable can take on at most $n+2$ values, so the number of program total states is bounded by: $m'(n+2)^{k'}$. This is also a polynomial in n, giving hope for Theorem 4.1 since the programs on either side of $\equiv$ have comparable numbers of states for a given input.

4.2 LOGSPACE *simulation of read-only programs*

Lemma 4.2 $\mathtt{WHILE}^{\backslash \mathtt{cons}} \preceq \mathtt{TM}^{logspace}$ □

Proof Let q = 1:I1 2:I2 ... m:Im be the GOTO version of a read-only WHILE program. Its instructions are of the form X := nil, X := Y, X := hd Y, X := tl Y, or if X goto ℓ else ℓ'.

The input to q is a list $(\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n) \in \mathbb{D}_{\mathtt{01}}$ corresponding to string $\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n$ in $\{\mathtt{0},\mathtt{1}\}^*$. Possible variable values in any q state have been analyzed above. Construct an off-line Turing machine p to simulate q, as follows. [7] The idea is to represent each variable X of q by its *position* and its *tag:* numbers (p_X, t_X) with values $(i, 0)$ in case (i), and $(0, 1)$ in case (ii), and $(0, 0)$ in case (iii). Turing machine program p stores each pair (p_X, t_X) on its work tape in binary,

[6] A value of nil can arise 3 ways, but the effects while executing p are the same: either it is the coding of some $a_i = \mathtt{0}$; or the nil at the end of the input list; or it is the head or tail of $c(\mathtt{1}) = (\mathtt{nil.nil})$.
[7] We assume $n > 0$; special case code gives the correct answer for $n = 0$.

thus using at most $O(\log n)$ bits for all of q's variables. GOTO command I is simulated by Turing commands achieving the following effects:

GOTO command I	Effect of corresponding Turing code
X := nil	$(p_X, t_X) := (0, 0)$
X := Y	$(p_X, t_X) := (p_Y, t_Y)$
X := hd Y	$(p_X, t_X) :=$ if $p_Y > 0 \wedge \mathtt{a}_{p_Y} = 1$ then$(0, 1)$else$(0, 0)$
X := tl Y	$(p_X, t_X) :=$ if $p_Y > 0$ then $(p_Y - 1, t_Y)$ else $(0, 0)$
if X goto ℓ else ℓ'	if $t_X = 1$ or $(p_X > 0 \wedge \mathtt{a}_{p_X} = 1)$ then goto ℓ else ℓ'

All is straightforward Turing programming; the test "$\mathtt{a}_{p_Y} = 1$" is done by scanning the Turing input tape $\mathtt{a}_1\mathtt{a}_1 \ldots \mathtt{a}_n$ left to right, until p_Y symbols have been seen. It is easy to see that this is a faithful simulation that preserves the representation of the variables in GOTO program q. □

4.3 Counter machines

The converse proof is a bit more subtle, as $O(\log n)$ bits of Turing machine work storage must be encoded using WHILE variables whose values are positions on the input string, and which can only be advanced forward (X := tl X). This will be done using the *counter machine* CM as an intermediate computation model, in the following stages ($\mathtt{CM}^{poly}$ and $\mathtt{CM}^{\backslash +}$ are defined below):

$$\mathtt{WHILE}^{\backslash \mathtt{cons}} \preceq \mathtt{TM}^{logspace} \preceq \mathtt{CM}^{poly} \preceq \mathtt{CM}^{\backslash +} \preceq \mathtt{WHILE}^{\backslash \mathtt{cons}}$$

Definition 4.3 A program in the language CM of *counter machines* is a sequence of labeled instructions p = 1:I1 2:I2 ... m:Im of the following forms for $0 <$ i and $0 \leq$ j (so counter C0 is never assigned).

```
I  ::=  Ci := Ci + 1 | Ci := Ci - 1 | Ci := Cj
     |  if Cj=0 goto ℓ else ℓ' | if Inᶜʲ = 0 goto ℓ else ℓ'
```

Storage has form $(\mathtt{d}, \sigma) \in \{0,1\}^* \times (I\!N \to I\!N)$ where d is the input data (read-only) and $\sigma(i)$ is the current contents of counter Ci for any $i \in I\!N$. Input to a counter machine is a string d in $\{0,1\}^*$. Data initialization sets counter C0 to n, giving the program a way to "know" how long its input is. The counter values σ are initialized to zero except for C0: initially,

$$\sigma_0(\mathtt{d}) = [0 \mapsto |\mathtt{d}|, 1 \mapsto 0, 2 \mapsto 0, \ldots]$$

A state for input d has form $s = (\ell, (\mathtt{d}, \sigma))$, where ℓ is the instruction counter. Input access is by instruction if $\mathtt{In}_{\mathtt{Ci}} = 0$ goto ℓ else ℓ'. Its effect: If $1 \leq \sigma(i) \leq n$ and $\mathtt{a}_{\sigma(i)} = 0$ then control is transferred to $\mathtt{I}_\ell$, else to $\mathtt{I}_{\ell'}$. The

remaining instructions behave as expected from the syntax, except that we define $0 - 1 = 0$ to avoid negative integers.

Thus p defines a one-step transition relation $s \to s'$. A *computation* on input $\mathtt{d} \in \{\mathtt{0},\mathtt{1}\}^*$ is a sequence $s_0 \to s_1 \to \ldots \to s_n$ where each s_i yields s_{i+1}, and s_0 equals $(1,(\mathtt{d},\sigma_0(\mathtt{d})))$, and s_n has instruction m+1 as first component (the "program end"). The output for input d is defined by:

$$[\![\mathtt{p}]\!](\mathtt{d}) = \begin{cases} \bot & \text{if p has no computation on input d, else} \\ \mathtt{1} & \text{if p's final state for input d has 1 in counter C1} \\ \mathtt{0} & \text{otherwise} \end{cases}$$

Since our aims concern only *decision powers* and not computation of functions, a counter machine output will be a single bit 0 or 1. Extension to outputs in $\{\mathtt{0},\mathtt{1}\}^*$ is routine, by a one-way write-only output tape. □

We now introduce some restrictions on counter machines.

Definition 4.4 A CM program p is $f(n)$-*bounded* if for any computation $s_0 \to s_1 \to \ldots \to s_n$ on input d, no counter in any state s_i exceeds $f(|\mathtt{d}|)$.

The language $\mathtt{CM}^{poly}$ is identical to CM, except that $\mathtt{CM}^{poly}$*-programs* is the subset of all CM programs p that are $f(n)$-bounded for some polynomial $f(n)$.

The language $\mathtt{CM}^{\backslash +}$ is identical to CM, except that $\mathtt{CM}^{\backslash +}$ programs contains no instructions of form Ci := Ci+1. □

Note that $\mathtt{CM}^{\backslash +}$ program is a $\mathtt{CM}^{poly}$ program. In fact, it is n-bounded, since all counters must remain less than or equal to the input length n.

4.4 Read-only simulation of LOGSPACE *programs*

Lemma 4.5 $\mathtt{TM}^{logspace} \preceq \mathtt{CM}^{poly}$ □

Proof Let p be a TM program that uses space at most $k \log n$ where n is the length of its input. We will show how to simulate p by a polynomially bounded counter machine. A total state of p is $s = (\ell, \mathtt{L}_1\underline{\mathtt{S}}_1\mathtt{R}_1, \mathtt{L}_2\underline{\mathtt{S}}_2\mathtt{R}_2)$ where ℓ is the instruction counter. The input tape is read-only and identical to the counter machine's input, so the contents of tape 1 need not be represented by counter s; its scanning position is sufficient.

The simulation represents p's work tape contents by two polynomially bounded counters, and simulates operations on either tape by corresponding counter operations. The scanning positions on tapes 1, 2 can be represented by counters

no larger than $n+2$ or $k \log n$, respectively. Both are certainly polynomially bounded. A work tape containing $\mathtt{b}_1 \ldots \underline{\mathtt{b}_i} \ldots \mathtt{b}_m$ can be represented by a pair of numbers l, r, where

- l is the value of $\mathtt{b}_1 \ldots \mathtt{b}_i$ as a base 3 number (counting `B` as digit 0, `0` as digit 1, and `1` as digit 2), and
- r is the value of $\mathtt{b}_m \mathtt{b}_{m-1} \ldots \mathtt{b}_{i+1}$ (note the reversal), also as a base 3 number.

The work tapes are initially all blank, so $l = r = 0$ at the simulated computation's start. Since $m \leq k \log n$, we have $l, r \leq 3^{k \log n} = n^{k \log 3}$. Thus altogether we have two counters to represent the input and work tape scanning positions, and two counters to represent the work tape contents. These counters are all bounded in size by $n+2$ or $n^{k \log 3}$, and collectively represent the Turing machine's total state. Each Turing machine operation can be faithfully simulated by operations on counters. For example, the effect of moving a work tape head right one position is to replace l by $3 \cdot l + (r \bmod 3)$, and to divide r by 3. It is easy to see that these operations can be done by counters. Testing the scanned square's contents amounts to a test on $l \bmod 3$, also easily done. □

Lemma 4.6 $\mathtt{CM}^{poly} \preceq \mathtt{CM}^{\backslash +}$ □

Proof

Counters bounded by n: Recall that counter `C0` is initialized to the length n of the input, and never re-assigned. We first show that any n-bounded `CM` program `p` is equivalent to some program `q` without `C:=C+1`. All counters are by assumption bounded by n, so we need not account for "overflow." First, the effect of `D := n - C` can be realized by the following:

```
D := C0; E := C; while E ≠ 0 do {E := E-1; D := D-1};
```

We can now simulate `C:=C+1` by:

```
D := n - C; D := D - 1; C := n - D
```

Polynomial-bounded programs: Now let `p` be an n^b-bounded `CM` program (extension to a general polynomial is straightforward). We show how to replace any variable `C` in `p` by a collection of $b+1$ variables, each bounded by n. The idea is simple: consider the base $n+1$ representation $k_b \ldots k_1 k_0$ of a number $k \leq n^b$ and represent `C` by $b+1$ register variables $\mathtt{C}_b, \ldots, \mathtt{C}_1, \mathtt{C}_0$.

Using this, the familiar algorithms for base $n+1$ addition and subtraction are easy to program, using `C0` to to recognize when "overflow" and "borrowing" occur. The test `if` $\mathtt{in}_\mathtt{C}$ `= 0 goto` ℓ `else` ℓ' is realized by transferring to ℓ' if any of $\mathtt{C}_b, \ldots, \mathtt{C}_1$ are nonzero, and otherwise performing `if` $\mathtt{in}_{\mathtt{C}0}$ `= 0 goto` ℓ `else` ℓ'. □

Lemma 4.7 $\mathtt{CM}^{\backslash +} \preceq \mathtt{WHILE}^{\backslash \mathtt{cons}}$ □

Proof An arbitrary $\mathtt{CM}^{\backslash +}$ program `p` must be shown equivalent to some read-only `WHILE` program. We do this by constructing a simulating program in `GOTO` form. Input to `p` is a string $\mathtt{a}_1\mathtt{a}_2...\mathtt{a}_n$, corresponding to `WHILE` input list $(\mathtt{a}'_1\mathtt{a}'_2...\mathtt{a}'_n) \in \mathbb{D}_{01}$. Each $\mathtt{a}'_i$ is `nil` if $\mathtt{a}_i = \mathtt{0}$, else `(nil.nil)`.

The counters `Ci` of `p` can only assume values between 0 and n. Represent each counter `Ci` by a corresponding `GOTO` program variable `Xi`, and represent the value of `Ci` by the distance from `Xi` to the end of the `WHILE` program's input. This gives the following simulation invariant:

Value of `Ci`	Value of `Xi`
0	`nil` $=$ `()`
$k > 0$	$(\mathtt{a}'_{n-k+1} \ldots \mathtt{a}'_n)$

Counter command `Ci := Cj` can obviously be simulated by `Xi := Xj`. Clearly `Ci` has value 0 if and only if `Xi` has value `nil`. Thus command `if Ci = 0 goto` ℓ `else` ℓ' can be simulated by `if Xi goto` ℓ' `else` ℓ. Further, instruction `Ci:=Ci-1` can be realized by `X := tl X`, reducing the distance to the end by 1. (This works for value 0 since the head of `nil` is `nil`.)

If `Ci` has value k then command `if` $\mathtt{in}_{\mathtt{Ci}}$ `= 0 goto` ℓ `else` ℓ' tests symbol $\mathtt{a}_k$. It can be simulated by the following (informal) code:

```
if Xi = nil goto ℓ;  (* if Ci = 0 goto ℓ *)
Z := tl Xi; Y := X0; (*  Y := (a'_1...a'_n) *)
while Z do           (*  At the loop end, Y = (a'_k...a'_n)  *)
   { Y := tl Y; Z := tl Z };
if hd Y goto ℓ' else ℓ
```

If `Ci` has value k then `Xi` has value $(\mathtt{a}'_{n-k+1} \ldots \mathtt{a}'_n)$, with k items. The while loop removes $k-1$ items from the start of input copy `Y` (initial value $(\mathtt{a}'_1\mathtt{a}'_2 \ldots \mathtt{a}'_n)$), leaving `Y` with value $(\mathtt{a}'_k\mathtt{a}'_{k+1} \ldots \mathtt{a}'_n)$. The head of this is `nil` if and only if $\mathtt{a}_k = \mathtt{0}$. □

Proof of Theorem 4.1. Lemmas 4.2, 4.7, 4.6, and 4.5 establish:

$$\mathtt{WHILE}^{\backslash \mathtt{cons}} \preceq \mathtt{TM}^{logspace} \preceq \mathtt{CM}^{poly} \preceq \mathtt{CM}^{\backslash +} \preceq \mathtt{WHILE}^{\backslash \mathtt{cons}}$$ □

5 Simulating recursive read-only programs in polynomial time

The main result of the rest of the paper is the following:

Theorem 5.1 $\texttt{WHILE}^{rec\setminus\texttt{cons}} \equiv \texttt{WHILE}^{ptime}$ □

This section establishes inclusion one way: that problems solvable by recursive read-only programs can be decided in polynomial time.

Implicit storage: the recursion stack.

Definition 5.2 Suppose `L` is the programming language `WHILE`, `CM`, or one of these restricted to a subset of programs (for instance, $\texttt{CM}^{poly}$). Thus any `L`-program has form `p = 1:I1 2:I2 ... m:Im`.

The *recursive extension* $\texttt{L}^{rec}$ is defined so $\texttt{L}^{rec}$*-programs* consists of all programs with syntax as in Figure 5.1, where each instruction `In`, `Jn` or `Kn` can be either:

- `call Pr` for some procedure `Pr`; or
- Any `L`-instruction, with two restrictions to avoid cross-procedural data references or `goto`s. First: in each procedure `Pi`, any referenced variable `X` must satisfy $\texttt{X} \in \{\texttt{U1}, \ldots, \texttt{Uu}, \texttt{Pi1}, \texttt{Pi2}, \ldots\}$, i.e. it must be either local or global. Second, the label ℓ in instruction `goto` ℓ refers to the procedure containing the instruction.

```
globalvariables U1,...,Uu;

procedure P1; localvariables P11,...,P1v;
    1:I1 2:I2 ... i:Ii i+1:

procedure P2; localvariables P21,...,P2w;
    1:J1 2:J2 ... j:Jj j+1:
   .....
procedure Pm; localvariables Pm1,...,Pmx;
    1:K1 2:K2 ... k:Kk k+1:

read U1; 1:call P1; 2: write U1
```

Figure 5.1: Recursive program syntax.

A *store* σ is a binding of variables to values. It is *global* if it binds `U1`,...,`Uu`, and local if it binds the local variables of some procedure `Pi`. A *total state* is

a stack (with the topmost element on the left) of form.[8]

$$(\ell_0, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \texttt{exit})$$

Control: each ℓ_i is a label either in `p`'s "main program" or in one of its procedures. Label ℓ_0 labels the instruction about to be executed (perhaps a procedure return), ℓ_1,...,ℓ_n are return addresses, and `exit` indicates program termination.

Storage: the last store σ_n always contains the global variable bindings (values of `U1`,...`Uu`), and σ_0 contains the local variable bindings of the most recently called procedure. The other $\sigma_0, \ldots, \sigma_{n-1}$ contain local variable values of earlier procedures that have been called but not yet returned from. Variable fetches and assignments are done using only σ_n and σ_0.

Semantics: this is as usual for imperative languages with recursion, and so is only briefly described. The *initial state* for input `d` is

$$(\texttt{1}, [\texttt{U1} \mapsto \texttt{d}, \texttt{U2} \mapsto \mathrm{DV}, \ldots, \texttt{Uu} \mapsto \mathrm{DV}], \texttt{exit})$$

where DV is an appropriate default value (`nil` for a tree value, or 0 for a counter value). Instruction "ℓ_0`: call P`" pushes a new stack frame in place of ℓ_0, yielding

$$(\texttt{1}, \sigma_{new}, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \texttt{exit})$$

Here `1` is the new procedure's initial control point, and σ_{new} assigns default values to all of `P`'s local variables. Thus label ℓ_0+`1` plays the role of "return address" (or `exit` for the initial call.)

When a procedure's last instruction has been executed, the leftmost label and store ℓ_0, σ_0 are popped off, and control is transferred to the instruction whose label is on the stack top. This yields total state

$$(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \texttt{exit})$$

□

[8] In compiler jargon this is the familiar "call stack." Each σ_i is a "stack frame" containing the local variable values of the currently called procedure. Label ℓ_i for $i > 1$ is the "return address," to which control will be passed after the current procedure's execution terminates.

Memoization: using explicit storage to simulate implicit storage. Suppose we are given an $\mathtt{L}^{rec}$ program p as in Figure 5.1. We outline informally a *memoizing simulation* of p on input d. The idea is that it records results of procedure calls as they are simulated. The underlying principle (due to Cook [3]) is that the net effect of a call to procedure P (if it terminates) is to transform the values in global store σ_n into a new global store σ'_n.

If procedure P is called a second time with the same global store σ_n, it will, of course, have the same effect; so the simulator can take a "short cut," directly changing its global store σ_n into σ'_n without bothering to simulate P again. This can lead to major time savings.

Construction. While the individual instructions of p are being simulated, a table *TAB* of triples of form $(\mathtt{P}, \sigma, \sigma')$ will be built (*TAB* stands for a "tabulation" of call effects). Entry $(\mathtt{P}, \sigma, \sigma') \in TAB$ signifies that procedure P has been called from a state with global store σ, and that it terminated with global store σ'.

Thus *TAB* will be used in two ways: to "archive" a triple $(\mathtt{P}, \sigma, \sigma')$ whenever a fact $\mathtt{P}(\sigma) = \sigma'$ has been established; and to retrieve from the archive the final effect of a procedure call which has earlier been simulated.

Program p is then simulated as follows: First, set $TAB = \{\}$ and initialize the total state (as above) to $(\ell_0, \sigma_0, \mathtt{exit})$ where $\sigma_0 = [\mathtt{U1} \mapsto \mathtt{d}, \mathtt{U2} \mapsto \mathrm{DV}, \ldots, \mathtt{Uu} \mapsto \mathrm{DV}]$. Then repeat the following until termination occurs (if it does) on the basis of the current total state

$$(\ell_0, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \mathtt{exit})$$

(i) *Non-call*: ℓ_0 labels a non-call instruction I. Then perform I as usual for L programs. This may increment or change ℓ_0, and change the local or global stores σ_0, σ_n.

(ii) *First call*: ℓ_0 labels an instruction `call P` and *TAB* contains no triple $(\mathtt{P}, \sigma_n, \sigma)$. Then change the total state to the following.[9] Note that execution of P's body will be done exactly as in the standard semantics; the first difference will appear on returning from P.

$$(\mathtt{1}, \sigma_{new}, \mathtt{1}, \sigma_n, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \mathtt{exit})$$

(iii) *Later call (short-cut)*: ℓ_0 labels an instruction `call P` and $(\mathtt{P}, \sigma_n, \sigma'_n) \in TAB$. Then change the total state to

$$(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma'_n, \mathtt{exit})$$

[9] This is exactly as for a normal call except for the components $\mathtt{1}, \sigma_n$: an extra copy of the global store has been added, with dummy label 1 to preserve the format.

and continue the simulation.

(iv) *Procedure return*: ℓ_0 is at the end of procedure P, and the total state is of form

$$(\ell_0, \sigma_{new}, 1, \sigma, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma', \texttt{exit})$$

Then add triple $(\texttt{P}, \sigma, \sigma')$ to *TAB*, change the total state to the following, and continue the simulation. Note the the overall effect of a call is exactly the same in both the original and the simulated versions.

$$(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \texttt{exit})$$

In summary: when simulating a first call to P, the above saves a copy of the global store on the stack top. On return from a first call simulation, the previous global store σ and the final global store σ' after the call are both available (σ was saved in step ii above). Using these, the triple $(\texttt{P}, \sigma, \sigma')$ is tabulated.

The same technique can be applied to nondeterministic programs; but table *TAB* then becomes a relation rather than a function, and the proof below needs modification.

Lemma 5.3 Suppose $\texttt{L}^{rec}$ program p terminates on all inputs, and enters at most $f(|\texttt{d}|)$ different local or global stores during the computation of $[\![\texttt{p}]\!](\texttt{d})$. Then the simulation described above of p on d can be carried out in time $O(f(|\texttt{d}|)^2)$. □

Proof First, the table *TAB* is initially empty, and an entry is added each time a procedure return is processed. This table can contain at most $O(f(|\texttt{d}|)^2)$ entries, since each consists of a procedure name (constant with respect to d) and two stores. [10]

Further, no entry $(\texttt{P}, \sigma, \sigma')$ is ever added twice to *TAB*. To see this, consider the first time procedure P is called with global store σ. We assumed program p terminates, so the call to P will exit; and P is not called again with the same store σ *within this call* (else it would have looped). Consequently a triple $(\texttt{P}, \sigma, \sigma')$ will be added to *TAB*, for the first time, on the first exit from P. If any subsequent call to P occurs with the same global store, the "short-cut" will be taken without effect on *TAB*.

Consequently *no* p *instruction is ever simulated twice with the same global store*, so the total number of instruction simulations carried out is proportional to the number of global stores: $O(f(|\texttt{d}|))$. For each the maximum time needed is the time to search or update *TAB*, which can be done in time $O(f(|\texttt{d}|))$

[10] The number is actually $O(f(|\texttt{d}|))$, since σ determines σ'.

(in fact, in substantially less time). In any case, the total simulation time is $O(f(|\mathtt{d}|)^2)$. □

PTIME **simulation of recursive read-only programs.**

Lemma 5.4 $\mathtt{WHILE}^{rec\backslash\mathtt{cons}} \preceq \mathtt{WHILE}^{ptime}$ □

Proof Let `p` be a $\mathtt{WHILE}^{rec\backslash\mathtt{cons}}$ program, and `d` its input with length n. Since `p` is `cons`-free, the value that any `p` store σ assigns to any variable `X` must be a pointer to some part of `d`, and thus a number between 0 and n. If `p` contains at most k variables, the number of stores on input `d` is at most $(n+1)^k$.

By Lemma 5.3, `p` can be simulated in time $O((n+1)^{2k})$, using the informal algorithm sketched there. Completion of the proof involves a programming of that algorithm in the `WHILE` language. This can clearly be done without increasing time by more than a small polynomial. □

6 Realizing polynomial time by recursive cons-free programs

The remaining goal is to show that any polynomial-time decidable subset of $\{\mathtt{0},\mathtt{1}\}^*$ can also be decided by a recursive cons-free `WHILE` program. This in effect shows that storage allocation by `cons` may be replaced by recursion. The proof will require a more specific storage model than seen earlier. Once this is set up, we will use a counter machine, as in the characterization of LOGSPACE, but in a recursive version.

Explicit storage: cons. We have assumed every elementary operation `cons`, `hd`, etc. as well as every conditional to take one time unit. These costs may seem illogical and even unreasonable since, for example, the command `X := cons Y Y` binds to `X` a tree with more than twice as many nodes as the one bound to `Y`. In fact, it *is* reasonable to assign constant time to a `cons` operation and the others using the *data-sharing implementation techniques* common to Lisp and other functional languages. In this section we give such a semantics for the flow chart version `GOTO`, using a Pascal-like simulation.

First, an example: consider the `reverse` program of Section 3.1.3. The two DAGs (directed acyclic graph) of Figure 6.1 illustrate the storage state at the beginning and at the end of execution.

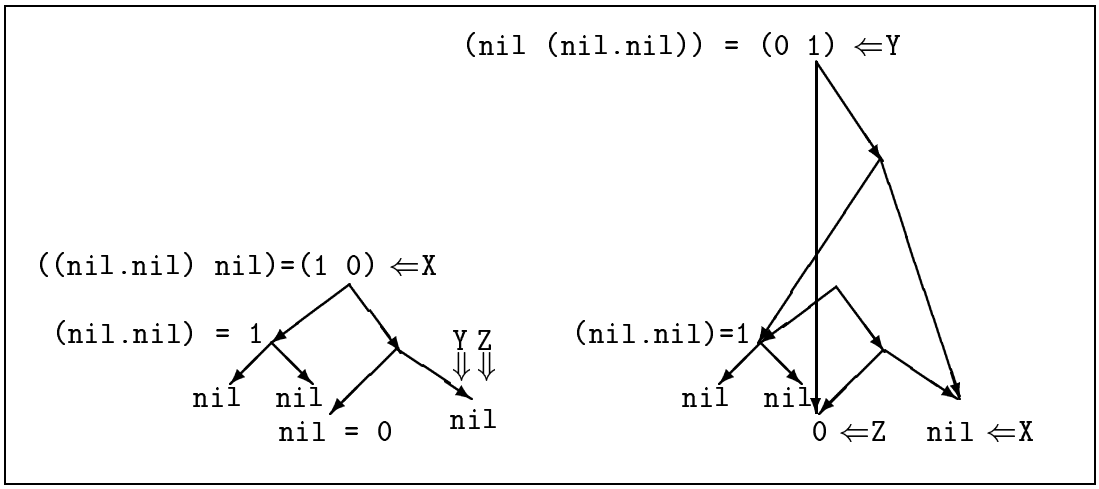


Figure 6.1: First and last DAG in execution of `reverse`

Construction. Let `p` = `1:`$\mathtt{I}_1$`; ...; m:`$\mathtt{I}_m$ be a `GOTO` program, with variables `X,Z1...,Zk` and with input and output through variable `X`. Construct a Pascal-like simulating program as follows:

```
type Index = 1..infinity;
     Node  =  0..infinity; (* 0 encodes nil *)
var  X, Y, Z1, ..., Zk : Node;
     Hd, Tl : array Index of Node;
       Time : Index;       (* The current step number    *)
Hd[0] := 0; Tl[0] := 0;    (* hd and tl of nil give nil  *)
X := 0; Z1:=0; ...; Zn:=0; (* Initialize all vars to nil *)
Time := 1;                 (* Step number initially 1    *)
1   : I1;           (* Code simulating p's instructions  *)
2   : I2;
...
m   : Im;
m+1 : writeout;            (* Write answer X using Hd, Tl *)
```

The storage is regarded as a DAG (initially empty for simplicity; input will be treated shortly). Command `Jl`, which simulates command $\mathtt{I}_l$ for $l \geq 0$, is defined in Figure 6.2.

The variable `Time` will keep track of the number of steps executed since the computation started, and so is zero when computation begins. The two parallel arrays `Hd`, `Tl` hold all pointers to `hd` and `tl` substructures. The values of variables `X1`, etc. will always be pointers to nodes in this DAG. A variable has value 0 if it is bound to `nil`, and otherwise points to a position in the `Hd` and `Tl` arrays.

For simplicity we handle allocation by using variable `Time` to find an unused

Form of $\mathtt{I}_l$	Simulating command `l:Jl`
`X := nil`	`X := 0; Time := Time + 1`
`X := Y`	`X := Y; Time := Time + 1`
`X := hd Y`	`X := Hd[Y]; Time := Time + 1`
`X := tl Y`	`X := Tl[Y]; Time := Time + 1`
`X := cons Y Z`	`Hd(Time) := Y; Tl(Time) := Z;` `X := Time; Time := Time + 1;`
`if X = nil goto r else s`	`if X = 0 then goto r else goto s`

Figure 6.2: Pascal-like program simulation code.

index in these arrays, so every DAG node will be identified by *the time at which it was created.*[11] Note that each of the simulation sequences in Figure 6.2 takes constant time, under the usual assumptions about Pascal program execution.

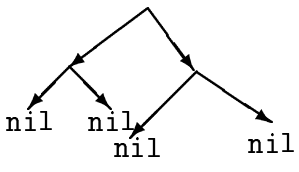
Suppose now that program `p` has input `d` $= (\mathtt{a}_1\ \mathtt{a}_2 \ldots \mathtt{a}_n) \in \mathbb{D}_{01}$. This data has to be stored into the Pascal data structures `Hd`, `Tl`. One way to describe this is to assume that variable `X` has been initialized by the following sequence of $n+3$ instructions, inserted at the start of `p`. Here `Zero` indicates the atom `nil`, modeled by the always-present cell 0:

`One := cons Zero Zero; X := Zero;` $\mathtt{Init}_1; \ldots \mathtt{Init}_n;$

Instruction $\mathtt{Init}_i$ for $1 \leq i \leq n$ is:

`X := cons Zero X` if $\mathtt{a}_i = 1$, else
`X := cons One X` if $\mathtt{a}_i = 1$

The following indicates the initial DAG built this way for input `d` $=$ `(1 0)`, coded as `((nil.nil) nil)`.



`Hd[0] = Tl[0] = 0`: Head, tail of `nil` = `nil`
`Hd[1] = Tl[1] = 0`: Cell for `One` = `(nil.nil)`
`X` = `nil` at start, no values in `Hd[2]` or `Tl[2]`
`Hd[3] = Tl[3] = 0`: cons `0` onto `X`
`X = 4, Hd[4] = 1, Tl[4] = 3`: cons `1` onto `X`

Trace of an example execution. Consider the `reverse` program seen before, and assume that it is given input `X = (1 0)` represented as above. This gives rise to the sequence of memory images in Figure 6.3, where

[11] By this model, the number of DAG nodes created during execution is at most the program's running time. A more parsimonious implementation could, for example, maintain a "free list" of unused memory cells.

$\texttt{Instr}_t$	=	the instruction about to be executed at time t
$\texttt{U}_t$	=	the DAG cell variable U is bound to at time t
$\texttt{Hd}_t$, $\texttt{Tl}_t$	=	the final values of Hd[t], Tl[t], respectively

Cons-free simulation of PTIME programs.

Lemma 6.1 Let $\texttt{p} = \texttt{1:I}_1\ \texttt{2:I}_2\ \ldots\texttt{m:I}_m$ be a GOTO-program, and let $\texttt{d} \in \mathbb{D}_{01}$ be an input. Let $(\ell_1, \sigma_1) \to \ldots \to (\ell_t, \sigma_t) \to \ldots$ be the (finite or infinite) computation of p on d, where $\ell_1 = 1$ and σ_1 is the initial DAG for input d. Then for any $t \geq |\texttt{d}| + 2$ and variable X the equations in Figure 6.4 hold. □

Proof A simple induction on t, using the code definitions from Figure 6.2.

Time t	$\texttt{Instr}_t$	$\texttt{Hd}_t$	$\texttt{Tl}_t$	$\texttt{X}_t$	$\texttt{Y}_t$	$\texttt{Z}_t$
0		0	0	-	0	0
1	Initialize data at	0	0	-	0	0
2	times	-	-	-	0	0
3	$t = 1, \ldots, n+2$	0	0	3	0	0
4		1	3	4	0	0
5	`1: Y := nil`	-	-	4	0	0
6	`2: if X goto 4`	-	-	4	0	0
7	`4: Z := hd X`	-	-	4	0	1
8	`5: Y := cons Z Y`	1	0	4	8	1
9	`6: X := tl X`	-	-	3	8	1
10	`7: goto 2`	-	-	3	8	1
11	`2: if X goto 4`	-	-	3	8	1
12	`4: Z := hd X`	-	-	3	8	0
13	`5: Y := cons Z Y`	0	8	3	13	0
14	`6: X := tl X`	-	-	0	13	0
15	`7: goto 2`	-	-	0	13	0
16	`2: if X goto 4`	-	-	0	13	0
17	`3: goto 8`	-	-	0	13	0
18	`8: write Y`	-	-	0	13	0

Figure 6.3: Execution trace for $[\![\texttt{reverse}]\!]\texttt{((nil.nil) nil)}$.

Lemma 6.2 $\texttt{WHILE}^{ptime} \preceq \texttt{CM}^{rec-poly}$ □

Proof Suppose one is given a WHILE-program p that runs in time $|\texttt{d}|^k$ on input d, and that $n = |\texttt{d}|$. Consider the values of $\texttt{Instr}_t, \texttt{Hd}_t, \texttt{Tl}_t, \texttt{X}_t$. If they can be computed then the value of output variable X is available through $\texttt{X}_{n^k+n+3}$. Thus $\texttt{WHILE}^{ptime} \preceq \texttt{CM}^{rec-poly}$ if we can show that all of the values $\texttt{Instr}_t$, etc. can be computed by a recursive counter program whose counters are polynomially bounded.

$$\mathtt{Instr}_{t+1} = \begin{cases} \mathtt{l'}\colon \mathtt{I}_{\ell'} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ goto l'} \\ \mathtt{l'}\colon \mathtt{I}_{\ell'} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ if X goto l' else l''} \text{ and } \mathtt{X}_t \neq 0 \\ \mathtt{l''}\colon \mathtt{I}_{\ell''} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ if X goto l' else l''} \text{ and } \mathtt{X}_t = 0 \\ \mathtt{l}+1\colon \mathtt{I}_{\ell+1} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \mathtt{I}_\ell \text{ otherwise} \end{cases}$$

$$\mathtt{Hd}_{t+1} = \begin{cases} \mathtt{Y}_t & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := cons Y Z} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathtt{Tl}_{t+1} = \begin{cases} \mathtt{Z}_t & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := cons Y Z} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathtt{X}_{t+1} = \begin{cases} \mathtt{X}_t & \text{if } \mathtt{Instr}_t \neq \mathtt{l}\colon \texttt{ X := } \cdots \\ \mathtt{Y}_t & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := Y} \\ \mathtt{Hd}_{(\mathtt{Y}_t)} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := hd Y} \\ \mathtt{Tl}_{(\mathtt{Y}_t)} & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := tl Y} \\ t+1 & \text{if } \mathtt{Instr}_t = \mathtt{l}\colon \texttt{ X := cons Y Z} \end{cases}$$

Figure 6.4: Relations among the values of Hd, Tl, X *for* $t > |\mathtt{d}| + 3$.

Regard each equation in Figure 6.4 as a definition of a function of one variable t. This is always a nonnegative integer, between 0 and $|\mathtt{d}|^k + |\mathtt{d}| + 3$, where the addend $|\mathtt{d}| + 3$ accounts for the time to initialize the Hd, Tl tables to represent input d.

The various functions $\mathtt{Instr}_t, \mathtt{Hd}_t, \mathtt{Tl}_t, \mathtt{X}_t$ are computable by mutual recursion, at least down to $t = |\mathtt{d}|$. Further, the program input d uniquely (and simply) determines the values of $\mathtt{Hd}_t, \mathtt{Tl}_t$ for $t < |\mathrm{d}|$ (and the other variables are irrelevant). The calls all terminate, since in each call the value of argument t decreases. Further, t is bounded by p's running time on d, at most $|\mathtt{d}|^k + |\mathtt{d}| + 3$.

Combining these, it is straightforward to build a recursive counter machine program q to simulate p; and it has polynomial size bounds on its counters.

Lemma 6.3 $\mathtt{CM}^{rec-poly} \preceq \mathtt{CM}^{rec\backslash +}$ □

Lemma 6.4 $\mathtt{CM}^{rec\backslash +} \preceq \mathtt{WHILE}^{rec\backslash \mathtt{cons}}$ □

Proof Lemma 4.6 showed $\mathtt{CM}^{poly} \preceq \mathtt{CM}^{\backslash +}$, and Lemma 4.7 showed $\mathtt{CM}^{\backslash +} \preceq \mathtt{WHILE}^{\backslash \mathtt{cons}}$. The constructions for these two results can be applied to recursive programs as well.

Remark. The recursive counter machine's computation is easily seen to take exponential time, due to recomputing values many times. For example, $\mathtt{Instr}_t$

is recomputed again and again. Thus even though a polynomial-time problem is being solved, the solver is running in superpolynomial time.

Proof of Theorem 5.1. Lemmas 5.4, 6.2, 6.3, and 6.4 establish:

$$\mathtt{WHILE}^{rec\backslash \mathtt{cons}} \preceq \mathtt{WHILE}^{ptime} \preceq \mathtt{CM}^{rec-poly} \preceq \mathtt{CM}^{rec\backslash +} \preceq \mathtt{WHILE}^{rec\backslash \mathtt{cons}} \qquad \square$$

Further developments. Stephen Cook [3] proved analogous results in the framework of "auxiliary push-down automata." Further developments involving efficient memoization led to the result that any 2DPDA (*two-way deterministic pushdown automaton*) can be simulated in linear time ([5,11,1]).

7 Conclusions and open questions

The programming approach yields quite simple intrinsic characterizations, by restrictions on program syntax, of the well-known problem classes LOGSPACE and PTIME, without external imposition of space or time bounds.

An interesting open problem. These two characterizations give new insight into the role of persistent (as opposed to evanescent) storage, and put questions about tradeoffs between computation time and space into sharp relief. The results above can be interpreted as saying that, in the absence of "cons," recursive programs are capable of simulating imperative ones; but at a formidable cost in computing time (exponentially larger), since results computed earlier cannot be stored but must be recomputed. In essence, we have shown that the heap can be replaced by the stack, but at a very high time cost.

It is not known, however, *whether this cost is necessary.* Proving that it is necessary (as seems likely) would require proving that there exist problems which can be solved in small time with general storage, but which require large time when computed functionally. A simple but typical example would be to establish a nonlinear lower bound on the time that a one-tape, no-memory *two-way pushdown automaton* [5], [11] requires to solve some decision problem. One instance would be to prove that string matching must take superlinear time. We conjecture that this is true.

Related work. The "treeless" programs of Wadler [16] also involve a restricted use of data constructors, and a treeless program of type $\{\mathtt{0},\mathtt{1}\}^* \rightarrow$

$\{0,1\}$ is a "read-only" program in our sense. Consequently the programs output by the "treeless transformer" compute LOGSPACE functions, if restricted to this input-output type [12]. Presumably those of more general type also have a simple characterization in terms of LOGSPACE.

Several other works have given exact characterizations of complexity classes in terms of programming languages or other formal languages. Most are, however, rather more technical than the LOGSPACE and PTIME characterizations above. Meyer and Ritchie's "loop language" characterization of the elementary functions [14] has a similar approach to that of this paper, but at a (much) higher complexity level. Another early result was the discovery that spectra of first-order logic are identical with NEXPTIME [9], further developed actively as finite model theory, see Immerman [8] and many others. Girard, Scedrov and Scott gave a linear logic characterization of PTIME in [6]. Bellantoni and Cook [4], Leivant and Marion [13], and Hillebrand and Kanellakis [7] have characterized several complexity classes by variants of the lambda-calculus.

References

[1] N. Andersen and N. D. Jones, Generalizing Cook's construction to imperative stack programs, Lecture Notes in Computer Science 812, pp. 1-18, Springer-Verlag, 1994.

[2] Manuel Blum: A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*, vol. 14, no. 2, pp. 322–336, 1967.

[3] S. A. Cook, Characterizations of pushdown machines in terms of time-bounded computers, *Journal of the Association for Computing Machinery* vol. 18, pp. 4–18, 1971.

[4] S. Bellantoni, S. A. Cook, A new recursion-theoretic characterization of the polytime functions, *Proceedings 24th Symposium on the Theory of Computing*, pp. 283-293, ACM Press, 1992.

[5] S. A. Cook, Linear-time simulation of deterministic two-way pushdown automata, *Information Processing* (IFIP) 71, C.V. Freiman, (ed.), North-Holland, pp. 75–80, 1971.

[6] J.-Y. Girard, A. Scedrov, P. Scott, Bounded linear logic: a modular approach to polynomial time computability, *Theoretical Computer Science*, vol. 97, pp. 1–66, 1994.

[12] Wadler's requirement of right-hand side linearity probably implies it is a proper subset of LOGSPACE.

[7] G. Hillebrand, P. Kanellakis, On the expressive power of simply typed and let-polymorphic lambda calculi, *Logic in Computer Science*, IEEE Computer Society Press, pp. 253–263, 1996.

[8] N. Immerman, Relational queries computable in polynomial time, *Information and Computation*, vol. 68, pp. 86-104, 1986.

[9] N. D. Jones, A. Selman, Turing machines and the spectra of first-order formulae with equality, *Journal of Symbolic Logic*, vol. 39, no. 1, pp. 139–150, 1974.

[10] N. D. Jones, Space-bounded reducibility among combinatorial problems, *Journal of Computer and System Science*, vol. 11, pp. 68–85, 1975.

[11] N. D. Jones, A note on linear-time simulation of deterministic two-way pushdown automata, *Information Processing Letters* vol. 6, pp. 110-112, 1977.

[12] N. D. Jones, Computability and Complexity from a Programming Perspective, MIT Press, 1997.

[13] D. Leivant, J-Y. Marion, Lambda calculus characterizations of ptime, *Fundamenta Informaticae*, vol. 19, pp. 167–184, 1993.

[14] A. Meyer, D. Ritchie, A classification of the recursive functions, *Zeitschrift MLG*, vol. 18, pp. 71–82, 1972.

[15] H. Rogers: Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.

[16] P. Wadler, Deforestation: transforming programs to eliminate trees. *European Symposium On Programming* (ESOP). Lecture Notes in Computer Science 300, pp. 344-358, Nancy, France, Springer-Verlag, 1988.