Tree Equality is not computable without ’cons’, a new characterization of LOGSPACE

Guillaume Bonfante\textsuperscript{1,2}\textsuperscript{*}, Mohamed El-Aqqad\textsuperscript{3}, and Mathieu Hoyrup\textsuperscript{4}

\textsuperscript{1} Ecole des Mines de Nancy, Université de Lorraine, Nancy, France
\textsuperscript{2} Loria, B.P. 239, 54506 Villers-lès-Nancy, France
\textsuperscript{3} Mohamed.El-Aqqad@mines-nancy.org
\textsuperscript{4} INRIA, Villers-lès-Nancy, 54506, France.
\textsuperscript{guillaume.bonfante@loria.fr}
\textsuperscript{3} Mohamed.El-Aqqad@mines-nancy.org
\textsuperscript{4} Mathieu.Hoyrup@inria.fr

Abstract

It is known that LOGSPACE corresponds to those programs that can read the data without touching them. This idea was developed by Neil Jones who gives a characterization of LOGSPACE by means of an elementary imperative language working on binary trees. To relate computations over words to those over trees, Neil Jones encodes words by linear trees, i.e. lists. We explore what happens when words are encoded in complete trees and more generally the computation power of cons-free programs on arbitrary trees. We show that a) full equality is not computable, b) to get back LOGSPACE, one needs a ’next’ operator, that is a ’for’ loop and c) we show that the complexity class of problems decidable by cons-free programs when input words are encoded in complete trees is incomparable with usual subclasses of LOGSPACE, under complexity assumptions.

1 Introduction

In [7] Jones gave characterizations of LOGSPACE and PTIME by means of an elementary imperative language While working on binary trees. His characterization of LOGSPACE is an elegant formalization of the intuitive idea that in LOGSPACE computations one can read the input without modifying it. He shows that LOGSPACE is the class of problems decidable by read-only While programs, i.e. programs that do not use the tree constructor cons. He also characterizes PTIME as the class of problems decidable by cons-free programs with recursion.

Although While programs work on general binary trees, Jones’ characterization actually assumes very particular trees as inputs: a finite binary string is encoded as a list, i.e. is written at the leaves of a linear tree, whose height is linear in the size of the string. In this article we investigate the computation power of cons-free While programs on general binary trees, and in particular when finite binary strings are encoded in the leaves of a complete binary tree.

We prove that:

\textsuperscript{*}Thanks to ANR Elica – ANR-14-CE25-0005 for its support.
• Equality between arbitrary binary trees is not decidable by \texttt{cons}-free programs,
• Using the complete tree encoding, one can restore the characterizations of LOGSPACE and PTIME by adding to the language a \texttt{next} operator that moves a pointer to its immediate brother or cousin in a tree,
• Using the complete tree encoding, the class of problems decidable by \texttt{cons}-free programs is not contained in AC$^0$ and is not contained in the classes ACC$^0$, TC$^0$, NC$^1$ assuming they differ from LOGSPACE. This is a step toward separation arguments as proposed by Hofmann, Ramyaa and Schöpp in their seminal contribution [6].

![Figure 1: Encoding the string 010110 in a linear tree or a complete tree](image)

The proof that equality is not computable by \texttt{cons}-free programs relies on an analysis of the computation that is about the operational semantics of the language. We hope in the future to get an ultimate obstinacy property in the style of Colson [3].

In a second step, we wanted to get back the result by Jones about LOGSPACE in the present terms. For that sake, we introduce an alternative semantics to programs in \texttt{While} which is based on pointers.

Several extensions of the result of Jones have been proposed. Let us mention the one by the first author [2] which characterizes some form of limited recursion that are stable for LOGSPACE. In [5] (followed by [9]), Carvalho and Simonsen extend the framework to rewriting. They give some syntactical properties that extend [2].

Then, we come back to the characterization of PTIME by Jones which itself is related to an older result by Cook [4](see also [1]).

The article is organized as follows. In Section 2 we present the language \texttt{While} and its basic properties. In Section 3 we prove that tree equality is not computable by the restricted language \texttt{While$^\setminus$cons} of programs that do not use the tree constructor \texttt{cons}. In Section 4 we propose an extension \texttt{While$^\#$} of \texttt{While$^\setminus$cons} with a \texttt{next} operator and a pointer semantics, and show that this language characterizes LOGSPACE. In Section 5 we show that the class of problems decided by \texttt{While$^\setminus$cons} programs over complete trees is incomparable with subclassed of LOGSPACE such as AC$^0$, and ACC$^0$, TC$^0$, NC$^1$, assuming they do not coincide with LOGSPACE. We do this by showing that LOGSPACE problems are decidable by \texttt{cons}-free programs with using oracles.

1.1 Notations

An alphabet is a finite set $\Sigma$ of letters. Given an alphabet $\Sigma$, let $T_\Sigma$ be the set of binary trees with leaves in $\Sigma$, that is the smallest set containing $\Sigma$ and $(t_0 \cdot t_1)$ whenever $t_0, t_1 \in T_\Sigma$. The two functions $\pi_0$ and $\pi_1$ are the projections: $\pi_i(t_0 \cdot t_1) = t_i$ and $\pi_i(c) = c$ with $i \in \{0, 1\}$ and $c \in \Sigma$. 
The size of a tree \( t \) is denoted \(|t|\) and is defined by \(|c| = 1, c \in \Sigma, \) and \(|(t \cdot u)| = |t| + |u| + 1.\) Its height, denoted \(|t|\) is defined as \(|c| = 0\) and \(|t \cdot u| = \max(|t|, |u|) + 1.\)

A word \( w = a_1 \cdot a_2 \cdots a_k \) in \( \Sigma^* \) is linearly encoded in \( T_{\Sigma \cup \{\text{nil}\}} \) as \( w = (a_1 \cdot (a_2 \cdots (a_k \cdot \text{nil}) \cdots )) \) where \( \text{nil} \) is an atom used as an end-marker. The tree encoding \( \hat{w} \) of a word \( w \in \Sigma^* \) is the smallest perfectly balanced tree in \( T_{\Sigma \cup \{\text{nil}\}} \) such that leaves read from left to right form \( w, \) possibly extended by the padding symbol \( \text{nil}. \) Both encodings relate computations over words to the ones over trees.

Let \( \preceq \) be the subtree relation on \( T_{\Sigma}, \) that is the smallest order (reflexive-transitive relation) such that for all \( t, u \in T_{\Sigma}: t \preceq t, t \preceq (t \cdot u) \) and \( t \preceq (u \cdot t). \)

## 2 The language While

We present (for pedagogical reasons, a slight variant of) While, a generic imperative language introduced by Jones [8]. We suppose a given alphabet \( \Sigma \) contains two atoms 0, 1 which stand respectively for false and true. Moreover, we suppose given (a denumerable set of) variables \( \text{Var} \ni X_0, X_1, \ldots. \) In the following, \( X, Y \) serve as generic variables. The syntax of While is given by the following grammar:

\[
\begin{align*}
\text{Expressions} & \ni E, F : \ni X | t | \text{cons } E \ F | \text{hd } E | \text{tl } E | E == c \\
\text{Commands} & \ni C, D : \ni X := E \mid C ; D \mid \text{while } E \text{ do } C \\
\text{Programs} & \ni p : \ni \text{read } X_1, \ldots, X_n ; C \ ; \text{return } Y
\end{align*}
\]

where \( t \in T_{\Sigma} \) and \( c \in \Sigma. \) Variables \( X_1, \ldots, X_n \) in \( p \) are called input variables, the other ones are qualified as local variables.

### 2.1 Semantics of While

A configuration, next called a store, is a function \( \sigma : \text{Var} \rightarrow T_{\Sigma}. \) The set of stores is denoted \( \mathcal{S}_{\Sigma}, \) or shorter, \( \mathcal{S} \) when \( \Sigma \) is clear from the context. Given a configuration \( \sigma \in \mathcal{S}, \) a variable \( X \) and \( t \in T_{\Sigma}, \sigma[X \mapsto t] \) is the store equal to \( \sigma \) on all variables but \( X \) for which it is set equal to \( t. \)

The semantics of an expression \( E \) applied on a configuration \( \sigma \) is denoted \( \llbracket E \rrbracket \sigma \) and defined by the equations:

\[
\begin{align*}
\llbracket X \rrbracket \sigma &= \sigma(X) \\
\llbracket t \rrbracket \sigma &= t \\
\llbracket \text{hd } E \rrbracket \sigma &= \pi_1(\llbracket E \rrbracket \sigma) \\
\llbracket \text{tl } E \rrbracket \sigma &= \pi_2(\llbracket E \rrbracket \sigma) \\
\llbracket E == c \rrbracket \sigma &= 1 \text{ if } \llbracket E \rrbracket \sigma = c \\
&= 0 \text{ otherwise.}
\end{align*}
\]

To define the operational semantics on commands, we introduce the “command” \( \text{stop}, \) which corresponds to the command remaining when the execution has terminated. The operational semantics is a function \( \text{op}() : \text{Commands} \times \mathcal{S} \rightarrow (\text{Commands} \cup \{\text{stop}\}) \times \mathcal{S}, \) defined as follows:

\[
\begin{align*}
\text{op}(X := E, \sigma) &= (\text{stop}, \sigma[X \mapsto \llbracket E \rrbracket \sigma]) \\
\text{op}(C : D, \sigma) &= (C : D, \sigma') \quad \text{if } \text{op}(C, \sigma) = (C', \sigma') \text{ with } C' \neq \text{stop} \\
\text{op}(C : D, \sigma) &= (D, \sigma') \quad \text{if } \text{op}(C, \sigma) = (\text{stop}, \sigma') \\
\text{op}(\text{while } X \text{ do } C, \sigma) &= (\text{stop}, \sigma) \quad \text{if } \sigma(X) = 0 \\
\text{op}(\text{while } X \text{ do } C, \sigma) &= (C : \text{while } X \text{ do } C, \sigma) \quad \text{otherwise.}
\end{align*}
\]
One can define \( \text{op}^k(C, \sigma) \) by \( \text{op}^0(C, \sigma) = (C, \sigma) \) and \( \text{op}^{k+1}(C, \sigma) = \text{op}(C', \sigma') \) if \( \text{op}^k(C, \sigma) = (C', \sigma') \) with \( C' \neq \text{stop} \).

The program \( p \triangleq \text{read } X_1, \ldots, X_n ; C ; \text{return } Y \) computes the following function on trees in \( T_\Sigma \). Given \( t_1, \ldots, t_n \), in the initial configuration \( \sigma_0(t_1, \ldots, t_n) \), all variables are set to \text{nil}, except \( X_1, \ldots, X_n \) which are respectively set to \( t_1, \ldots, t_n \). Then, let \( k \) be the first index such that \( \text{op}^k(C, \sigma_0(t_1, \ldots, t_n)) = (\text{stop}, \sigma) \). We say that the computation is done in \( k \) steps. We define \( \text{J}_p(t_1, \ldots, t_n) = \sigma(Y) \).

From there, one computes functions over words via the linear or the tree encoding.

### 2.2 Syntactic sugar

Tests may use expressions rather than variables. For instance, for an expression \( E \)

\[
\text{while } E \text{ do } C \triangleq \begin{cases} 
Y := E; \\
\text{while } Y \text{ do } \{ C; Y := E \},
\end{cases}
\]

with \( Y \) a fresh variable. One may define a test statement

\[
\text{if}(E)\{C\}\text{else}\{D\} \triangleq \begin{cases} 
Z := 1; Y := E; \\
\text{while } Y \text{ do } \{ Y := 0; Z := 0; C \}; \\
\text{while } Z \text{ do } \{ Z := 0; D \},
\end{cases}
\]

where we suppose that \( Y \) and \( Z \) are fresh.

On the alphabet \( \{0, 1\} \), one may verify if a tree pointed by \( X \) is a leaf or not with

\[
\text{isLeaf}(X) \triangleq \begin{cases} 
\text{if}(X == 0)\{Z := 1\} \\
\text{else } \{\text{if}(X == 1)\{Z := 1\} \\
\text{else } \{Z := 0\}\},
\end{cases}
\]

the result being stored in \( Z \).

We can encode boolean values as well as boolean operators in the language: \text{true} is \( 1 \) and \text{false} is \( 0 \),

\[
-E \triangleq \text{if}(E)\{Z := \text{false}\} \text{ else } \{Z := \text{true}\},
\]

\[E \text{ and } F \triangleq \text{if}(E)\{Z := F\} \text{ else } \{Z := \text{false}\},
\]

\[E \text{ or } F \triangleq \text{if}(E)\{Z := \text{true}\} \text{ else } \{Z := F\}.
\]

Finally, one may define and use functions as long as they are not recursive. Functions are 'inlined', that is they are seen as macros with possibly a variable renaming in case there is a clash. If a program \( p \triangleq \text{read } X_1, \ldots, X_n ; C ; \text{return } Y \) has already been defined then one may use the expression \( p(E_1, \ldots, E_n) \) where \( E_1, \ldots, E_n \) are expressions. Any occurrence of \( p(E_1, \ldots, E_n) \) can be replaced by some fresh variable \( Y \) and preceded by the command \( X_1 := E_1; \ldots ; X_n := E_n ; C \), assuming that \( X_1, \ldots, X_n \) are fresh.

### 3 Equality is not computable without \text{cons}

Let \( \text{While}^{\cong} \subseteq \text{While} \) denote the set of such programs, that is without “\text{cons}” expressions. We restate the result of Jones in the present context:
Theorem 3.1 (Jones [7]). Problems computed by programs in While$^{\text{cons}}$ with linear encoding are exactly Logspace problems.

The main problem we want to address is to deal with computations on trees. Is there any analogous statement? The answer is no: even one of the simplest problems is not computable.

Theorem 3.2. Equality is not computable in While$^{\text{cons}}$.

It contrasts with Jones’ observation in his book [8] (p. 42, Section 2.3) that equality is computable in the full language While including cons (in the book, equality is even included as an atomic operation in the language, but can be tested using equality to nil).

We now prove the theorem. Let us first give an outline of the argument:

- In a complete uniform tree of height $h$, the number of subtrees is about $h$,
- So the number of possible stores during the execution of a program on that input is at most $h^k$ where $k$ is the number of variables of the program,
- So if the program terminates on that input, it must terminate in at most $h^k$ steps,
- But then it cannot “visit” every position in the tree, as there are about $n = 2^h$ positions,
- So we can change the value of some leaf in the tree so that the program execution will not be altered,
- But then the program cannot answer correctly on both the original and the modified trees.

We will formalize the idea of visited position through the notion of formal store: while a store assigns a value, i.e., a tree to each variable, a formal store assigns a position in the input tree to each variable.

Polynomial time.

Lemma 3.1. For each command $C$ there is a finite set $S$ of commands and a finite set $T$ of trees such that for each $\sigma$ and each $k$, if $\text{op}(C, \sigma) = (C', \sigma')$ then $C'$ belongs to $S$ and for each variable $X$, $\sigma'(X)$ belongs to $T$ or is a subtree of $\sigma(Y)$ for some variable $Y$.

Proof. $S$ can be easily defined by induction on $C$.

$T$ is the set of constant trees appearing as expressions in $C$, together with their subtrees and the atoms 0 and 1. For any assignment $X := E$ in $C$, $E \sigma$ either belongs to $T$ if $E$ contains no variable or is an equality test, or is a subtree of some $\sigma(Y)$ if $E$ is built from $Y$ by applications of $\text{hd}$ and $\text{tl}$.

Observe that the result can be iterated: if $\text{op}^k(C, \sigma) = (C', \sigma')$ then the same result holds: $C' \in S$ and each $\sigma'(X)$ belongs to $T$ or is a subtree of some $\sigma(Y)$.

A complete uniform tree is a complete tree whose leaves have the same value $c \in \Sigma$.

Proposition 3.1. If $C \in$ While$^{\text{cons}}$ then there exists a polynomial $P(h)$ such that if $\sigma$ contains only complete uniform trees of heights $\leq h$ and $C$ terminates on $\sigma$, then it terminates in at most $P(h)$ steps.

Proof. As the number of subtrees of a complete uniform tree of height $\leq h$ is polynomial in $h$, the number of possible values of $\text{op}^k(C, \sigma)$ for all $k$ is polynomial. If $C$ terminates on $\sigma$ then one must have $\text{op}^k(C, \sigma) \neq \text{op}^{k'}(C, \sigma)$ for $k \neq k'$, so the number of steps must be polynomial.
We now observe that an execution step of a command depends on the values of the store, but only superficially. More precisely, we say that two trees $t, t'$ are superficially equal if both $t$ and $t'$ are leaves with the same label ($t = t' \in \Sigma$) or both $t$ and $t'$ are not leaves (they have positive heights). Two stores $\sigma, \sigma'$ are superficially equal if for each variable $X$, $\sigma(X)$ is superficially equal to $\sigma'(X)$. If $\sigma$ and $\sigma'$ are superficially equal then they lead to the same execution step for each command $C$.

**Lemma 3.2.** If $\sigma$ and $\sigma'$ are superficially equal then $\text{op}(C, \sigma)$ and $\text{op}(C, \sigma')$ have the same first component.

**Proof.** The only situation where the next command depends on the store is when $C$ starts with *while* $X$ *do* $D$ for some $D$, and only one of $\sigma(X)$ and $\sigma'(X)$ is 0, which is not possible if $\sigma$ and $\sigma'$ are superficially equal. □

**Formal stores.**

We know from Proposition 3.1 that if the inputs of a program are complete uniform trees then the program has limited (i.e. polynomial) time to visit these trees. But it may happen that all their subtrees appear as values of variables during the execution. Indeed, many subtrees at different positions actually coincide. For instance if $X$ contains a complete uniform tree then the expressions $\text{hd } X$ and $\text{tl } X$ evaluate to the same tree. However these two expressions do not correspond to the same positions in the tree, which is indicated by the fact that they are not *formally* equal. We introduce the notion of a formal store, that enables one to distinguish them.

A formal store is a function $S : \text{Var} \to \text{Expressions}$. Observe that every store is also a formal store. Intuitively a formal store is a store that is not evaluated: the values given by the initial store are lacking.

A formal store $S : \text{Var} \to \text{Expressions}$ can be extended to a morphism $S : \text{Expressions} \to \text{Expressions}$, in the obvious way: $S(\text{hd } E) = \text{hd } S(E)$, $S(\text{tl } E) = \text{tl } S(E)$, $S(t) = t$, $S(E == c) = (S(E) == c)$.

Let $S$ be the set of formal stores. An execution step of a command $C$ on a formal store $S$ induces a new formal store $S'$ denoted by $\text{fs}(C, S)$. The function $\text{fs}() : \text{Commands} \times S \to S$ is defined as follows:

\[
\begin{align*}
\text{fs}(X := E, S) &= S[X \mapsto S(E)] \\
\text{fs}(C : D, S) &= \text{fs}(C, S) \\
\text{fs}(\text{while } X \text{ do } C, S) &= S.
\end{align*}
\]

Let $\sigma_0$ be an initial store. Let $C_0 = C$. As long as they are defined, let $(C_{k+1}, \sigma_{k+1}) = \text{op}(C_k, \sigma_k)$. Let $S_0$ be the initial formal store defined by $S_0(X) = X$ for input variables and $S_0(X) = \text{nil}$ for other variables, and let $S_{k+1} = \text{fs}(C_k, S_k).

As said before, the formal store $S_k$ is a store that is not evaluated, the missing part is given by the initial store $\sigma_0$. Indeed, $\sigma_k$ can be derived from $\sigma_0$ and $S_k$.

**Lemma 3.3.** For every expression $E$, $\llbracket E \rrbracket \sigma_k = \llbracket S_k(E) \rrbracket \sigma_0$.

**Proof.** As $\llbracket \cdot \rrbracket \sigma_k$ and $\llbracket S_k(\cdot) \rrbracket \sigma_0$ are morphisms, they coincide on $\text{Expressions}$ if and only if they coincide on $\text{Var}$.

We prove the result by induction on $k$. For $k = 0$, $\llbracket S_0(X) \rrbracket \sigma_0 = \llbracket X \rrbracket \sigma_0$ by definition of $S_0$. Now assume that $\llbracket \cdot \rrbracket \sigma_k = \llbracket S_k(\cdot) \rrbracket \sigma_0$ (and $C_k \neq \text{stop}$).
If \( C_k \) starts with \textbf{while} \( X \) \textbf{do} \( D \) then \( S_{k+1} = S_k \) and \( \sigma_{k+1} = \sigma_k \).
If \( C_k \) starts with \( X := E \) then \( \sigma_{k+1}(X) = [E] \sigma_k = [[S_k(E)]]\sigma_0 \) and \( S_{k+1}(X) = S_k(E) \).
The other variables are not affected.

Let \( \sigma_0 \) be such that for all \( X \in \text{Var} \) and all \( k, [S_k(X)]\sigma_0 \) and \( [[S_k(X)]]\sigma_0 \) are superficially equal. Let \( C_0, C_1, \ldots, \sigma_0, \sigma_1, \ldots \) and \( S_0, S_1, \ldots \) be the execution traces of \((C, \sigma_0)\).

**Lemma 3.4.** For all \( k \) one has \( C_k = C_k \), \( S_k = S_k \) and \( \sigma_k \) and \( \sigma_k \) are superficially equal.

**Proof.** We prove it by induction on \( k \).
First \( C_0 = C_0 = C \) and \( S_0 = S_0 \) is the identity.
Assume that \( C'_k = C_k \) and \( S'_k = S_k \). First, according to the definition of \( f_s() \), one has \( S'_{k+1} = S_{k+1} \). Then, since \( S'_k = S_k \), by Lemma 3.3, \( \sigma_k = [[S_k(X)]\sigma_0 \) and \( \sigma'_k = [[S_k(X)]\sigma'_0 \) Thus, \( \sigma_k \) and \( \sigma'_k \) are superficially equal by the choice of \( \sigma'_0 \). By Lemma 3.2, \( C'_{k+1} = C_{k+1} \).

**Lemma 3.5.** Let \( E \) be a finite set of expressions, \( \sigma \) a store and \( X \) a variable. If the number of leaves in the tree \( \sigma(X) \) is larger than the cardinality of \( E \) then there is a store \( \sigma' \) such that:

- \( \sigma'(Y) = \sigma(Y) \) for all \( Y \neq X \),
- \( \sigma'(X) \) is obtained by changing the value of one leaf in \( \sigma(X) \),
- \( [E]\sigma \) and \( [E]\sigma' \) are superficially equal for all \( E \in E \).

**Proof.** There is a leaf in \( \sigma(X) \) that no expression in \( E \) points to. Simply change the value of that leaf.

We now conclude the proof of Theorem 3.2. Let \( p \triangleq \text{read} \ X_1, X_2 ; C ; \text{return} \ Y \) be a program in \texttt{While} that computes equality: \( [p](t_1, t_2) \) evaluates to 1 if \( t_1 = t_2 \), to 0 otherwise. Let \( P(h) \) be a polynomial bounding the terminating time of \( p \) when the inputs \( t_1 = t_2 \) are complete uniform trees of height \( h \) (Proposition 3.1). Let \( h \) be such that \( 2^h > P(h) \), \( t_1 = t_2 \) complete uniform trees of height \( h \) and \( \sigma_0 \) the corresponding initial store mapping \( X_i \) to \( t_i \) \( (i = 1, 2) \). Let \( E \) be the set of expressions \( S_k(X) \) for \( X \in \text{Var} \) and \( k \in \mathbb{N} \), as long as they are defined. There are at most \( P(h) \) expressions in \( E \) \( (S_{k+1} \) differs from \( S_k \) at most at one variable) and there are \( 2^h > P(h) \) leaves in \( t_2 \) so one can change the value of one leaf of \( t_1 \) as in Lemma 3.5, let \( \sigma'_0 \) be the corresponding initial store and \( \sigma_k \) and \( \sigma'_k \) be the final stores after execution of \( p \) on \( \sigma_0 \) and \( \sigma'_0 \) respectively.

One has \( \sigma_k(Y) = [S_k(Y)]\sigma_0 \) which is superficially equal to \( \sigma'_k(Y) = [S_k(Y)]\sigma'_0 \) as \( S_k(Y) \in E \). However as \( p \) computes equality one must have \( \sigma_k(Y) = 1 \) and \( \sigma'_k(Y) = 0 \), contradiction.

### 4 A pointer semantics

As discussed in the previous section, cons-free programs have a strong pointer flavor. Second property, equality is not computable by means of the atomic one. In other words, there is no chance we can relate properly the class of problems computed by cons-free programs with any reasonable complexity classes. We propose a new extension of cons-free programs with two ingredients: a \texttt{next} operator which acts on pointers and a test \texttt{null}, again acting on pointers. We enrich expressions as follows.

\[
\text{Expressions } \ni E ::= X \mid t \mid \text{hd} \ E \mid t \cdot E \mid E == c \mid \text{next} \ E \mid \text{null} \ E
\]
where \( t \in T_\Sigma \) and \( c \in \Sigma \). Commands and programs are left unchanged. We call these programs pointer programs and we denote their set by \( \text{while}^p \). To keep a tractable notation, we dropped the \( \text{cons} \) index but these programs are implicitly \( \text{cons} \)-free.

Intuitively, the \texttt{next} operator applied on a node outputs the immediate next brother or cousin. The expressions \( \text{null} E \) evaluate to 0 or 1 depending on the fact that we are exploring a leftmost branch of an input (or a constant of the program). The functions \( \text{hd} \) and \( \text{tl} \) serve for depth traversal and \texttt{next} for width visits.

Let us consider the alphabet \( \Delta = \text{Var} \cup T_\Sigma \cup \{\text{hd}, \text{tl}\} \). We define \( \text{pointers} \) to be words within

\[
\text{Pointers} = (\text{hd} \mid \text{tl})^* \cdot (\text{Var} \cup T_\Sigma).
\]

A pointer corresponds to an expression: \( X = X, \ t = t, \ \text{hd} \cdot p = \text{hd} \ p \) and \( \text{tl} \cdot p = \text{tl} \ p \). In the following, we will identify pointers to their expressions.

A pointer store is a mapping \( \Pi : \text{Var} \rightarrow \text{Pointers} \). We denote by \( \mathcal{P} \) the set of pointer stores. We extend the pointer mapping to expressions relatively to a store \( \sigma \) by induction on the structure of the expression. First, \( \Pi_\sigma(t) = t \) and \( \Pi_\sigma(X) = X(\sigma) \). For an expression \( E \), let \( p = \Pi_\sigma(E) \). Then, \( \Pi_\sigma(\text{hd} \ E) = p \) if \( E \sigma \in \Sigma \), otherwise \( \Pi_\sigma(\text{hd} \ E) = \text{hd} \ p \). In a similar way, \( \Pi_\sigma(\text{tl} \ E) = p \) if \( E \sigma \in \Sigma \), otherwise \( \Pi_\sigma(\text{tl} \ E) = \text{tl} \ p \). Finally, \( \Pi_\sigma(E = c) = 1 \) if \( E \sigma = c \), otherwise \( \Pi_\sigma(E = c) = 0 \). For \texttt{next}, we apply the following transducer \( T_0 \) on the pointer:

Accordingly, we define \( \Pi_\sigma(\text{next} \ p) = \Pi_\sigma(T_0(p)) \). For \texttt{null}, we define \( \Pi_\sigma(\text{null} \ p) = 1 \) if \( p \in \text{hd}^* \cdot (\text{Var} \mid T_\Sigma) \) and \( \Pi_\sigma(\text{null} \ p) = 0 \) otherwise.

Given a store \( \sigma \), an execution step of a command \( C \) on a pointer store \( \Pi \) induces a new command \( C' \) and a new pointer store \( \Pi' \) denoted by \( \text{pop}_\sigma(C, \Pi) \). The function \( \text{pop}_\sigma() : \text{Commands} \times \mathcal{P} \rightarrow \text{Commands} \times \mathcal{P} \) is defined as follows:

\[
\begin{align*}
\text{pop}_\sigma(X := E, \Pi) &= (\text{stop}, \Pi[X \mapsto \Pi_\sigma(E)]) \\
\text{pop}_\sigma(C : D, \Pi) &= (C' : D, \Pi') & \text{if } \text{pop}_\sigma(C, \Pi) = (C', \Pi') \text{ with } C' \neq \text{stop} \\
\text{pop}_\sigma(C : D, \Pi) &= (D, \Pi') & \text{if } \text{pop}_\sigma(C, \Pi) = (\text{stop}, \Pi') \\
\text{pop}_\sigma(\text{while} X \text{ do } C, \Pi) &= (\text{stop}, \Pi) & \text{if } \llbracket \Pi(X) \rrbracket \sigma = 0 \\
\text{pop}_\sigma(\text{while} X \text{ do } C, \Pi) &= (C : \text{while} X \text{ do } C, \Pi) & \text{otherwise}.
\end{align*}
\]

We suppose given a program \( p \triangleq \text{read} \ x_1, \ldots, x_n ; \ C ; \text{return} \ y \) and values \( t_1, \ldots, t_n \in T_\Sigma \). Let \( \sigma_0(t_1, \ldots, t_n) \) be the initial store. Let \( C_0 = C \) and \( \Pi_0 \) map \( x_1, \ldots, x_n \) to themselves and the other variables to \texttt{null}. Then, we set the sequence \( (C_0, \Pi_0), (C_1, \Pi_1), \ldots \) as long as its elements are defined: suppose given \( (C_k, \Pi_k) \), then \( (C_{k+1}, \Pi_{k+1}) = \text{pop}_{\sigma_k}(C_k, \Pi_k) \).

Let \( k \) be the minimal index for which \( \sigma_k = \text{stop} \). We define the output of the program on \( t_1, \ldots, t_n \in T_\Sigma \) to be \( \llbracket p \rrbracket(t_1, \ldots, t_n) = \llbracket \Pi_k \rrbracket(y) \sigma_0 \).

Let us justify now that the semantics we defined in the previous section coincide with the present one on \texttt{while} \texttt{cons} programs. Let \( (C'_k, \sigma_k) \) be defined by means of \( \text{op}() \): \( C'_0 = C_0 \) and \( (C'_{k+1}, \sigma_{k+1}) = \text{op}(C'_k, \sigma_k) \).
Lemma 4.1. Let \( \sigma \) be a store such that \([X] \sigma = [\Pi_\sigma(X)] \sigma_0\). For any expression \( E \), we have \([E] \sigma = [\Pi_\sigma(E)] \sigma_0\).

Proof. By induction on \( E \).

Lemma 4.2. Suppose that we are given a pointer store \( \Pi \) and a store \( \sigma \) such that \([X] \sigma = [\Pi(X)] \sigma_0\) for all variables \( X \). Given a command \( C \), write \( (C', \Pi') = \text{pop}_{\sigma_0}(C, \Pi) \) and \( (C'', \sigma') = \text{op}(C, \sigma) \), then \( C' = C'' \) and for all variables \( X \), \([X] \sigma' = [\Pi'(X)] \sigma_0\).

Proof. By induction on \( C \).

- If \( C = (X := E) \), then \( C' = C'' = \text{stop} \) and for all \( Y \neq X \), we have \( [Y] \sigma' = [\Pi(Y)] \sigma_0 \) and \( [Y] \sigma'' = [\Pi(Y)] \sigma_0 \). Thus, \([Y] \sigma' = [Y] \sigma = [\Pi(Y)] \sigma_0 = [\Pi'(Y)] \sigma_0 \). For the variable \( X \), thanks to Lemma 4.1, \([X] \sigma' = [\Pi_\sigma(E)] \sigma_0 = [\Pi'(X)] \sigma_0\).

- For the sequential case, \( C = D \ ; \ D' \), it is immediate by induction.

- For \( C = \text{while} \ X \ \text{do} \ D \), observe that \( [X] \sigma = \Pi(X) \sigma_0 \). Thus, the two operations \( \text{op}() \) and \( \text{pop}_{\sigma_0}() \) share the same control. Thus, \( C'' = C' \). And since \( \sigma' = \sigma \) and \( \Pi' = \Pi \), we have \([X] \sigma' = [\Pi'(X)] \sigma_0\) for all variables \( X \).

Lemma 4.3. Given the definitions \((C_k, \Pi_k)_k\) and \((C'_k, \sigma_k)_k\) as above, for all \( k \), we have: \( C'_k = C_k \) and \([X] \sigma_k = [\Pi_k(X)] \sigma_0\) for all variables \( X \).

Proof. By induction on \( k \). For \( k = 0 \), this is by definition of \( C_0, C'_0, \Pi_0 \) and \( \sigma_0 \). The induction step is a direct consequence of Lemma 4.2.

Keeping notations above, the two operational semantics output the same result as a corollary of Lemma 4.3.

Corollary 4.1. Let \( C_k \) be the first index such that \( C_k = \text{stop} \). We have \([Y] \sigma_k = [\Pi_k(Y)] \sigma_0\).

A pointer \( p \) is well-formed if \( p = w_1 \cdot w_2 \) with \( w_1 \in (\text{hd} \ | \ \text{tl})^* \) and \( w_2 \in \text{Var} \cup T_\Sigma \) and \(|w_1| \leq |w_2|\). In other word, the word \( w_1 \) encodes a proper path within \( w_2 \). By extension, we say that a pointer store \( \Pi \) is well-formed if \([X] \pi\) is well-formed for each variable \( X \).

Lemma 4.4. Given a well-formed pointer store \( \Pi \), for any expression \( E \), any store \( \sigma \), and \( \Pi_\sigma(E) \) is well-formed.

Proof. By induction on the expression \( E \). For constants, \( E \equiv c \) and \( \text{null}_E \), the result being a constant, it is well-formed. For variables the result is immediate from the hypothesis. For \( \text{hd} \) and \( \text{tl} \), this is due to the test \([p] \sigma \in \Sigma \). For \( \text{next}(E) \), one observe that the transducer keeps the size of words unchanged (and does not change the ultimate letter in \( \text{Var} \cup T_\Sigma \)).

As corollary, we can state that in a computation, since the initial pointer store \( \Pi_0 = X_1 \rightarrow X_i, i = 1, \ldots, n \) is well-formed, any pointer store \( \Pi_k \) occuring within the computation are so.

4.1 Capturing Logspace

Theorem 4.1. Problems computed by programs in \( \text{While}^P \) with tree encoding are exactly Logspace problems.

We cut the proof into two propositions.

Proposition 4.1. Problems computed in Logspace are computable within \( \text{While}^P \).
The proof relies on the following theorem (see for instance Jones [8], § 21.3):

**Theorem 4.2.** \( \text{Logspace}^{TM} = \text{Logspace}^{CM} \).

That is problems computed within \( \text{Logspace} \) on a Turing Machine are exactly those computed in \( \text{Logspace} \) on a read-only counter machine. We recall that a counter machine is described as a sequence \( 1 : I_1, \ldots, k : I_k \) of instructions:

\[
I ::= C_i := C_i + 1 \mid C_i := C_i - 1 \mid C_i := C_j \mid \\
\quad \text{if } C_i = 0 \text{ goto } l \text{ else goto } l' \mid \text{if } \text{In}_{C_i} = 0 \text{ goto } l \text{ else goto } l'
\]

where \( i, j \) are integers and \( C_i, C_j \) are counters. At the beginning, each counter but \( C_0 \) is set to 0 and \( C_0 \) contains the input. Next, \( C_i \) denote the content of counter \( C_i \). The instructions speak for themselves. For the last one, the control is transferred to \( l \) or to \( l' \) depending on the fact that the \( C_i \)-th bit of \( C_0 \) is 0 or not.

**Proof.** Suppose we are given a read-only counter machine working in \( \text{Logspace} \) (that is the content of counters \( C_i \) is always below \( n \) with \( n \) the size of the input). We show that we can simulate it step after step by a While\(^P\) program.

The principle is the following. To each counter \( C_i \), \( i > 1 \), we associate a variable pointing to the leaf of the input (seen here as a tree) corresponding to the content of \( C_i \). We show that we can simulate any instruction of the machine.

- \( C_i := C_i + 1 \). We apply the instruction \( C_i := \text{next } C_i \). Since \( C_i < n \), we never do an "overflow", that is pointing on the left-most branch.
- \( C_i := C_i - 1 \). We apply the instruction \( C_i := \text{minusOne}(C_i, C_0) \) defined below.
- \( C_i := C_j \). We apply the instruction \( C_i := C_j \).
- \( \text{if } C_i = 0 \text{ goto } l \text{ else goto } l' \). The variable \( Y := \text{null } C_i \) serves for control transfer.
- \( \text{if } \text{In}_{C_i} = 0 \text{ goto } l \text{ else goto } l' \). The variable \( Y := C_i \) serves for control transfer.

For the control of the machine, we use an extra PC variable and we proceed by finite case analysis.

\[
\begin{align*}
\text{leftmost}(X) & \quad L := X; \\
& \quad \text{while}(\neg \text{isLeaf}(L)) \{ \\
& \quad \quad L := \text{hd}(L) \\
& \quad \} \\
& \quad \text{return } L
\end{align*}
\]

\[
\begin{align*}
\text{symmetric}(C, X) & \quad L := \text{leftmost}(X); \\
& \quad C_1 := C; \\
& \quad \text{while}(\neg \text{null } C_1) \{ \\
& \quad \quad C_1 := \text{next } C_1; \\
& \quad \quad L := \text{next } L; \\
& \quad \} \\
& \quad \text{return } L
\end{align*}
\]

\[
\begin{align*}
\text{minusOne}(C, X) & \quad C := \text{symmetric}(C, X); \\
& \quad C := \text{next } C; \\
& \quad C := \text{symmetric}(C, X); \\
& \quad \text{return } C
\end{align*}
\]

**Proposition 4.2.** Any function computed within While\(^P\) is computable in logarithmic space.

**Proof.** We suppose we are given a program \( p = (\text{read } X_1, \ldots, X_n; C; \text{return } Y) \) and values \( t_1, \ldots, t_n \in T_\Sigma \). Define \( \sigma_0 \) as above. To evaluate the code, we simulate the \( \text{pop}_{\sigma_0}(-) \) function, step after step.

Without loss of generality, we suppose that the variables of the program are \( X_1, \ldots, X_{m_0} \). Furthermore, we suppose that the program involves constant trees (the \( t \)'s) enumerated as \( t_{m_0+1}, \ldots, t_{m_1} \). Next, \( t_i \) denotes \( \sigma_0(X_i) \) for \( i \leq m_0 \).
Each pointer $\Pi(X_i)$, $i \leq m_0$ is represented by means of two working tape $T_{d,i}$ and $T_{p,i}$ ($d$ for data, $p$ for path). Suppose that $\Pi(X_i) = w_1 \cdot w_2$ with $w_1 \in (hd, tl)^*$ and $w_2 \in \{X_1, \ldots, X_{m_0}\} \cup \Sigma$. Set $T_{d,i} = j$ if $w_2 = X_j$ for some $j \leq m_0$, otherwise set $T_{d,i} = j$ where $j$ is the index of $t$ as defined above. We set $T_{p,i} = w_1$.

Let us evaluate the size of working tapes. The $T_{d,i}$'s contain a number within 1, $\ldots$, $m_1$, thus constant space. Any tape $T_{p,i}, i \leq m_0$, contains a word, say $w_i$. Due to Lemma 4.4, this word $w_i$ verify $|w_i| \leq ||t_{T_{d,i}}||$. Since the $t_i$'s are either inputs or constants, the size of $|w_i|$ takes logarithmic space.

The program counter management is done by finite case analysis. Thus, the result.

4.2 Polynomial time

We end the section, restoring the characterization of polynomial time of Neil Jones within the current context. Now, programs may be recursive. A program is defined to be a finite list of procedures, one of which is called main. Each procedure is defined: $\text{Name}(\text{localvariables}) \text{C}$ where $\text{Name}$ denotes the name of the function, local variables are set to nil each time the procedure is called and $\text{C}$ is a command extended to the new statement call $P$ where $P$ is a procedure name. Procedures interact through global variables that occur within their code. Such programs are denoted by $\text{recWhile}^p$. We refer to Jones [8] for a full presentation.

Theorem 4.3. Problems computed in PTIME are exactly those computable within $\text{recWhile}^p$ with tree encoding.

Proof sketch (cf. Appendix). We begin to prove that programs within $\text{recWhile}^p$ are computable in polynomial time. The argument is essentially the one of Jones. We use memoization. We define a table which associate to each procedure a map $(\text{Var} \rightarrow \text{Pointers}) \rightarrow (\text{Var} \rightarrow \text{Pointers})$. Each time a procedure is called, it skips the computation if it is present in the table. Notice that the size of such a map is polynomial with respect to the size of inputs. As before, we count the number of configurations, there are only polynomially many, thus the result.

In the other direction, based on the fact that we showed how to emulate LOGSPACE in the last section, it is tedious but not difficult to prove that we can simulate a machine working in alternating logarithmic space.

5 Capturing LOGSPACE using oracles

In this section we come back to the language $\text{While}^{\text{cons}}$ and compare the corresponding complexity class with other subclasses of LOGSPACE.

We know that the simplest problems such as equality, or having only zeroes, are not decidable by programs in $\text{While}^{\text{cons}}$. Hence this class of programs does not contain any reasonable complexity class below LOGSPACE. We prove here that there is no simple upper bound either on their complexity, other than LOGSPACE.

Theorem 5.1. Let $\mathcal{C} \subseteq \text{LOGSPACE}$ be a complexity class among $\{\text{AC}^0, \text{ACC}^0, \text{TC}^0\}$.

Under the assumption $\mathcal{C} \neq \text{LOGSPACE}$, the complexity class of $\text{While}^{\text{cons}}$ is not contained in $\mathcal{C}$.

In particular, the complexity class is not contained in $\text{AC}^0$.

In order to prove the theorem, we show that a complete tree can be entirely explored using $\text{While}^{\text{cons}}$ programs with the help of an oracle, and more generally that any problem in LOGSPACE can be decided by a program in $\text{While}^{\text{cons}}$ using an oracle. We show two types
of oracles that allow such computations: oracles whose height is exponential in the height of
the input, and oracles with the same height as the input. The idea underlying the first ones is
easier to understand and is part of the mechanism of the second class of oracles. We then show
at the end of the section how these oracle computations can be used to prove Theorem 5.1.

5.1 Deep oracles
First, it is possible to explore the whole content of a complete tree of height \( n \) if we are given
another tree whose height is exponential in \( n \). Let \( T \) be a complete tree of height \( n \) and \( H \) a tree
of height \( 2^n - 1 \). We show here a program that computes the parity on the leaves of \( T \), using \( H \)
as oracle. Similar programs can be defined to compute any problem in \( \text{Logspace} \).

The idea is that any path leading to a leaf in \( T \) can be represented as a number between 0
and \( 2^n - 1 \). As a result, if \( H \) has height \( 2^n - 1 \), then the leaves of \( T \) can be entirely explored
by iteratively exploring the leaf coded by the height of \( H \) and decrementing the height of \( H \) (by
replacing \( H \) with \( \text{hd}(H) \)).

The left-height of a tree \( t \) is the height of its leftmost path (number of heads) and is denoted
by \( ||t||_{\text{left}} \). One can write the following programs in \( \text{While} \)

- The program \( \text{isOdd}(H) \) decides whether \( ||H||_{\text{left}} \) is odd.
- The program \( \text{half}(H) \) returns a tree whose left-height is \( ||H||_{\text{left}} \) divided by 2 (quotient of
  the Euclidean division).
- The program \( \text{getBit}(H,K) \) returns the value of bit \( k \) in the binary expansion of \( ||H||_{\text{left}} \),
  where \( k = ||K||_{\text{left}} \) (bit 0 is the rightmost bit).
- The program \( \text{getLeaf}(T,H) \) returns the node of \( T \) whose path is given by the rightmost \( n \)
  bits in the binary expansion of \( ||H||_{\text{left}} \), assuming that \( T \) is complete of height \( n \). Contrary
to what could be expected, here 0 codes for \( \text{tl} \) and 1 codes for \( \text{hd} \). Indeed in the sequel
we will start with \( ||H||_{\text{left}} = 2^n - 1 \), so its binary expansion contains only 1’s, and we want
to explore \( T \) starting from the leftmost path, so 1 should correspond to \( \text{hd} \).

If \( T \) is a complete tree of height \( n \) and \( H \) has left-height \( 2^n - 1 \) then one can read all the leaves
of \( T \) and perform computations with them. We show a generic program reading the leaves of \( T \),
as well as an example program computing the parity:

\[
\begin{align*}
generic(T, H) & \\
\text{stop} & := \text{false}; \\
\text{while}(\neg \text{stop}) \{ \\
\quad L & := \text{getLeaf}(T, H); \\
\quad \// \text{do something with } L \\
\quad \text{stop} & := \text{isLeaf}(H); \\
\quad H & := \text{hd}(H); \\
\} \\
\end{align*}
\]

\[
\begin{align*}
\text{parity}(T, H) & \\
R & := \text{true}; \\
\text{stop} & := \text{false}; \\
\text{while}(\neg \text{stop}) \{ \\
\quad L & := \text{getLeaf}(T, H); \\
\quad \text{if}(L=1) \\
\quad R & := \neg R; \\
\quad \text{stop} & := \text{isLeaf}(H); \\
\quad H & := \text{hd}(H); \\
\} \\
\text{return } R
\end{align*}
\]

We give other examples that will be used in the sequel.

- The program \( \text{onlyZero}(T,H) \) tells whether all the leaves of \( T \) are 0’s, assuming that \( T \)
is complete of height \( n \) and \( H \) has left-height \( 2^n - 1 \).
The program \texttt{followPath}(T, X, G, H) returns the subtree of \(X\) whose path is given by the leftmost \(|G|\)-left leaves of \(T\), assuming that \(T\) is a complete tree of height \(n\) and \(H\) has left-height \(2^n - 1\). This time, 0 codes for \(\texttt{hd}\) and 1 codes for \(\texttt{tl}\). This specification might look awkward, but will be used by the next type of oracles.

### 5.2 Counting oracles

We now show that a complete tree of height \(n\) can be completely explored using as oracle another complete tree \(C_n\) of height \(n\) with special values at the leaves, called a \textit{counting oracle} and obtained as follows. Given \(n\), we define

\[ p = \log_2(n) := 1 + \lfloor \log(n + 1) \rfloor. \]

One has in particular \(2^p \geq n - p\). Let \(w_n\) be the string of length \(2^n\) obtained by enumerating the binary strings of length \(n - p\) in lexicographic order starting from \(0^n - p\) up to \(1^n - p\) and then \(0^{n-p}\), add padding 0’s at the end of each one to have length \(2^p\), and concatenating them. For instance with \(n = 6\), we have \(p = 3\) and \(w_n\) is:

\[001000001000001100000100000001010000011000000000000000000000000000000000\]

**Definition 5.1.** We define the \textbf{counting oracle} of height \(n\), denoted \(C_n\), as the complete binary tree of height \(n\) with \(w_n\) written at its leaves.

Let us explain how the \textit{counting oracle} \(C_n\) can be entirely explored, and then be used to explore any other tree of height \(n\):

- Any subtree of \(C_n\) of height \(p = \log_2(n)\) can be entirely explored using \(C_n\) as a deep oracle like in the previous section,
- Each subtree of \(C_n\) of height \(p\) encodes in its leaves the path leading to the next subtree of height \(p\).

As a result, one can explore all the leaves of \(C_n\) by starting with the leftmost subtree \(S\) of height \(p\) and iterating the following procedure: read the leaves of \(S\) using \(C_n\) as a deep oracle, follow in \(C_n\) the path encoded by the leaves of \(S\); this leads to the next subtree of height \(p\) which we then store in \(S\). Stop when the leaves of \(S\) are all 0.

Let us now be more precise. In order to prove that any \textsc{Logspace} problem can be solved by a program in \texttt{While\textsc{cons}} using a counting oracle, we follow the argument used to prove Proposition 4.1. We have to show how pointers and the operator \texttt{next} can be simulated.

**Simulating pointers.** A pointer to a leaf of a complete tree \(T\) of height \(n\) is just a binary string of length \(n\). It will be decomposed into three strings of lengths \(n - p\), 1 and \(p - 1\) respectively, stored in several variables in different ways:

- The variable \(T1\) contains the subtree of height \(p\) the leaf belongs to, and at the same time the variable \(P1\) contains the subtree of \(C_n\) at the same position as \(T1\) in \(T\). This represents the \(n - p\) first symbols of the pointer to the leaf.
- The second part is just a bit stored in \(P2\).
- The third part will be encoded as the left-height of a tree of left-height between 0 and \(2^{p-1} - 1\), stored in \(P3\). We use the convention of the program \texttt{getLeaf}: the left-height of \(P3\) in binary codes a path by replacing 1 with \(\texttt{hd}\) and 0 with \(\texttt{tl}\).

At the beginning the pointer positioned at the leftmost leaf is represented in the following way: \(T1\) and \(P1\) point to the leftmost subtrees of height \(p\) in \(T\) and \(C_n\) respectively, \(P2\) is 0 and \(P3\) is \(H\), a tree whose left-height is \(2^{p-1} - 1\) (such an \(H\) can be obtained from \(C_n\) as \(n \geq 2^{p-1} - 1\)).
Simulating the next operator. Applying the next operator is simulated as follows:

- If $||P3||_{\text{left}} > 0$ then decrement $||P3||_{\text{left}}$ (by replacing $P3$ with $\text{hd}(P3)$),
- Otherwise let $P3 = H$. If $P2$ is 0 then replace it with 1,
- Otherwise let $P2 = 0$. Read the leaves of $P1$ using $H$ as a deep oracle and follow the corresponding path in $C_n$ and $T$ to update $P1$ and $T1$ respectively. This is possible by calling the program $\text{followPath}$, as $P1$ has height $p - 1$ and $H$ has height $2^{p-1} - 1$.

The programs. We now list the programs in While\text{\texttt{cons}} that are used to implement these ideas.

- The program $\text{minus}(A, B)$ returns the leftmost subtree of $A$ whose left-height is $\max(0, ||A||_{\text{left}} - ||B||_{\text{left}})$.
- The program $\text{minusLogg}(H)$ returns the leftmost subtree of $H$ at depth $\logg(||H||_{\text{left}})$.
- The program $\text{logg}(H)$ returns the leftmost subtree of $H$ whose left-height is $\logg(||H||_{\text{left}})$.
- The program $\text{nextSubtree}(S, T, H)$ returns the subtree of $T$ whose path is given by the leftmost $n - p$ leaves of $S$, assuming that $T$ is a complete tree of height $n$, $S$ is a complete tree of height $p = \logg(n)$ and $H$ has left-height $2^{p-1} - 1$. This program makes calls to $\text{followPath}$ and 0 codes for $\text{hd}$ and 1 codes for $\text{tl}$.

A Mersenne number is a number in the shape $2^k - 1$ for some natural number $k \geq 1$. The Mersenne numbers are the numbers whose binary expansions contain only 1’s.

- The program $\text{isMersenne}(H)$ decides whether $||H||_{\text{left}}$ is a Mersenne number.
- The program $\text{mersenne}(H)$ returns a tree whose left-height is the greatest Mersenne number below $n = ||H||_{\text{left}}$, i.e. $2^{p-1} - 1$ where $p = \logg(n)$.

We now come with the pointers implementation explained above. To each pointer to a leaf of $T$ correspond variables $T1$, $P1$, $P2$ and $P3$. For simplicity, we assume that they are global and are modified by calls to the following programs. The variable $H$ is also global and will contain a tree of height $2^{p-1} - 1$ obtained from $C_n$ at the very beginning.

- The program $\text{init}(T, C)$ initializes the variables $T1$, $P1$, $P2$ and $P3$ so that they simulate a pointer to the leftmost leaf of $T$.
- The program $\text{isEnd}()$ returns true if the current pointed leaf is the rightmost one.
- The program $\text{next}()$ modifies $T1$, $P1$, $P2$ and $P3$ so that they represent the next leaf.
- The program $\text{getCurrentLeaf}()$ returns the current pointed leaf.

We give the generic program for reading all the leaves of $T$ given the oracle $C$ (the counting tree with the same height as $T$), and an example program that computes parity.
More generally, every problem in \textsc{Logspace} can be solved a program in \textsc{While}\textsubscript{cons} using counting oracles, by simulating pointers as explained above.

5.3 Proof of the theorem

Using similar arguments, we can show that there is a program \( q \) in \textsc{While}\textsubscript{cons} such that \( q(C) \) decides whether \( C \) is a counting oracle: at each step it reads a subtree \( S \) of \( C \) of height \( p = \log^g(n) \), goes to the subtree \( S' \) whose path is encoded by the leaves of \( S \) and checks that the leaves of \( S' \) encode the successor of the leaves of \( S \), and continues with \( S' \) in place of \( S \).

Proof of Theorem 5.1. Let \( P \) be a problem in \textsc{Logspace}\textbackslash C. We consider the following problem: given \((C,T)\), decide whether \( C \) is a counting oracle and \( T \in P \). We show that this problem is decidable by a program in \textsc{While}\textsubscript{cons} and is not in \( C \).

There is a program \( p \) in \textsc{While}\textsubscript{cons} such that \( p(T,C) \) decides \( T \in P \) when \( C \) is a counting oracle. Let \( q \) be a program in \textsc{While}\textsubscript{cons} such that \( q(C) \) decides whether \( C \) is a counting oracle. Using \( p \) and \( q \) one can easily build a program \( r(T,C) \) that decides whether \( C \) is a counting oracle and whether \( T \in P \). If this problem is in \( C \) then there is a family of circuits deciding this problem. Specializing these circuits with a counting oracle in place of the input \( C \) gives a family of circuits in the same class, deciding \( P \). This is impossible as \( P \notin C \) by assumption, so the problem decided by \( r \) is not in \( C \).

\[ \square \]

References


A Proof of Theorem 4.3

\textbf{recWhile}^p \textbf{is in Ptime} \hspace{1em} \text{We give some details about the proof of Theorem 4.3. To prove that computation in \textbf{recWhile}^p are done in polynomial time, we use the following strategy. First, for each procedure } P \text{, we associate a dictionary mapping } (X_1, \ldots, X_k \rightarrow \text{Pointers}) \rightarrow (X_1, \ldots, X_k \rightarrow \text{Pointers}) \text{ where } X_1, \ldots, X_k \text{ are the global variables. Second, we use a call stack.}

\text{First observation, given preceding discussions, it is clear that given some inputs of size } n \text{, there are } p(n) \text{ different pointers for some polynomial } p \text{. Thus, the dictionary has at most polynomial size, say } q(n). \hspace{1em}

\text{Second observation, the depth of the stack can be limited to polynomial depth. Indeed, the stack has the shape } (P_1, \mathbf{p}_1), (P_2, \mathbf{p}_2), \ldots, (P_m, \mathbf{p}_m) \text{ with } P_i \text{ some procedure names and } \mathbf{p}_i \text{ the pointers corresponding to the global variables.}

\text{The result of the evaluation of a procedure depends only on the value of global variables. There are only polynomially many configurations. Thus, each time one has to evaluate a procedure } P \text{ on some values } \mathbf{p} \text{ one looks at the table and if the configuration } \mathbf{p} \text{ is already present, one skips the recursive evaluation and updates directly the global variables according to the table. Now, since the program is not looping, the size of the stack is bounded by } |P| \times q(n), \text{ that is a polynomial.}

\text{Conclusion, the stack has polynomial size, each step is done in polynomial time (it is in LOGSPACE!), thus, the result.}

\textbf{Ptime is in recWhile}^p \hspace{1em} \text{In the other direction, we suppose given a problem } \mathcal{P} \text{ computable in polynomial time. Then, we can suppose it is computed by an alternating Turing Machine } \mathcal{M} \text{ working in logarithmic space. We simulate such a machine in } \textbf{recWhile}^p.

\text{First fact. We can initialize local variables of recursive calls in } \textbf{recWhile}^p. \text{ Consider for instance that } P_1 \text{ calls } P_2:

\begin{center}
\begin{tabular}{|c|}
\hline
\text{P1(X,Y)} \\
\hline
\text{C ;} \\
\text{Z := E ;} \\
\text{call P2 ;} \\
\hline
\end{tabular}
\begin{tabular}{|c|}
\hline
\text{P2(T,U,Y)} \\
\hline
\text{T := Z ;} \\
\text{C ;} \\
\hline
\end{tabular}
\end{center}

\text{where } Z \text{ is a fresh global variable. The computation of } P_2 \text{ starts with a local variable } T \text{ set to } E. \text{ Such a call is summed up call } P_2(E) \text{ in the following.}

\text{The machine } \mathcal{M} \text{ works in logarithmic space, that is stacks in the computation have logarithmic size. Let } \mathcal{M} = (Q, q_0, \delta). \text{ A state } q \in Q \text{ is either a computing node labeled } \wedge
of \lor, or a reading state or an accepting state or a rejecting state. The initial configuration is \((q_0, \ldots)\) with \(k\) stacks. Updating a stack is performed as we did in Section 4.1. Say 
\[\text{Update}(Q, S_1, \ldots, S_k, a_1, \ldots, a_k)\]
updates the stacks \(S_1, \ldots, S_k\) according to the actions \(a_1, \ldots, a_k\) (increment, decrement or reset). Let the following program:

\begin{verbatim}
Simulate(Q, S_1, ..., S_k)
  if (Q is accepting) { Result := 1 }
  else { if (Q is rejecting) { Result := 0 }
          else { if (Q is reading) { Result := S_1
          if (Q is computing) {
            Update(Q, S_1, ..., S_k, delta(., 0))
            Res_1 := Result
            Update(Q, S_1, ..., S_k, delta(., 1))
            Res_2 := Result
            Result := Res_1 op Res_2; }
          } }
  }
\end{verbatim}

In the program, \(\delta(., 0)\) describes the actions for the left branch the computing tree and \(\delta(., 0)\) for the right ones. The variable \(\text{Result}\) is global (and stores the result), the other ones are local. In the expression \(\text{Res}_1 \text{ op } \text{Res}_2\), \(\text{op}\) denotes \(\land\) or \(\lor\) according to the state. Since all stacks are limited to logarithmic size, the updates are correctly performed. Thus the simulation.

B Programs codes

Here we give the codes of the programs mentioned in Section 5.

B.1 Deep oracles

We give the programs of Section 5.1.

- The program \(\text{isOdd}(H)\) decides whether \(\left| H \right|_{\text{left}}\) is odd.
- The program \(\text{half}(H)\) returns a tree whose left-height is \(\left| H \right|_{\text{left}}\) divided by 2 (quotient of the Euclidean division).
- The program \(\text{getBit}(H, K)\) returns the value of bit \(k\) in the binary expansion of \(\left| H \right|_{\text{left}}\), where \(k = \left| K \right|_{\text{left}}\) (bit 0 is the rightmost bit).
- The program \(\text{getLeaf}(T, H)\) returns the node of \(T\) whose path is given by the rightmost \(n\) bits in the binary expansion of \(\left| H \right|_{\text{left}}\), assuming that \(T\) is complete of height \(n\). Contrary to what could be expected, here 0 codes for \(t_l\) and 1 codes for \(h_d\). Indeed in the sequel we will start with \(\left| H \right|_{\text{left}} = 2^n - 1\), so its binary expansion contains only 1's, and we want to explore \(T\) starting from the leftmost path, so 1 should correspond to \(h_d\). Note that the variables of the programs are local, i.e. calling a program from another program does not affect the variables of the outer program (they should be renamed when merging programs).
• The program \texttt{parity}(T, H) returns the parity for the leaves of T, assuming that T is complete of height $n$ and H has left-height $2^n - 1$.

• The program \texttt{onlyZero}(T, H) tells whether all the leaves of T are 0’s, assuming that T is complete of height $n$ and H has left-height $2^n - 1$.

• The program \texttt{followPath}(T, X, G, H) returns the subtree of X whose path is given by the leftmost $||G||_{\text{left}}$ leaves of T, assuming that T is a complete tree of height $n$ and H has left-height $2^n - 1$. This time, 0 codes for \texttt{hd} and 1 codes for \texttt{tl}. This specification might look awkward, but will be used by the next type of oracles.

\textbf{B.2 Counting oracles}

We give the programs of Section 5.2

• The program \texttt{minus}(A, B) returns the leftmost subtree of A whose left-height is $\max(0, ||A||_{\text{left}} - ||B||_{\text{left}})$.

• The program \texttt{minusLogg}(H) returns the leftmost subtree of H at depth $\log_{2}(|H|_{\text{left}})$.

• The program \texttt{logg}(H) returns the leftmost subtree of H whose left-height is $\log_{2}(|H|_{\text{left}})$. 

18
The program `nextSubtree(S, T, H)` returns the subtree of `T` whose path is given by the leftmost `n − p` leaves of `S`, assuming that `T` is a complete tree of height `n`, `S` is a complete tree of height `p = logg(n)` and `H` has left-height `2^p − 1 − 1`. This program makes calls to `followPath` and 0 codes for `hd` and 1 codes for `tl`.

As `H` has height `2^p − 1 − 1`, it can be used as a deep oracle to explore a complete tree of height `p − 1`. Hence `S` can be explored by exploring both `hd(S)` and `tl(S)`.

A Mersenne number is a number in the shape `2^k − 1` for some natural number `k ≥ 1`. The Mersenne numbers are the numbers whose binary expansions contain only 1’s.

- The program `isMersenne(H)` that decides whether `|H|_{left}` is a Mersenne number.
- The program `mersenne(H)` returns a tree whose left-height is the greatest Mersenne number below `n = |H|_{left}`, i.e. `2^p − 1 − 1` where `p = logg(n)`.

The variables `H`, `T1`, `P1`, `P2` and `P3` are global and are modified by calls to the following programs.

- The program `init(T, C)` initializes the variables `P1`, `P2` and `P3` so that they encode the path of the leftmost leaf of `T`. It also initializes `T1` so that it points, in `T`, to the same position as `P1` in `C`.
- The program `isEnd()` returns true if the current leaf is the rightmost one.
• The program `next()` modifies $T_1$, $P_1$, $P_2$ and $P_3$ so that they represent the next leaf.

• The program `getCurrentLeaf()` returns the current leaf.

```
<table>
<thead>
<tr>
<th>init(T, C)</th>
<th>isEnd()</th>
<th>getCurrentLeaf()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H := \text{mersenne}(C);$</td>
<td>$\text{R := !hasNextSubtree}(P_1, H);$</td>
<td>if ($P_2 = 0$)</td>
</tr>
<tr>
<td>$T_1 := \log g(T);$</td>
<td>and $P_2 = 1;$</td>
<td>$L := \text{hd}(T_1);$</td>
</tr>
<tr>
<td>$P_1 := \log g(C);$</td>
<td>and $\text{isLeaf}(P_3);$</td>
<td>else</td>
</tr>
<tr>
<td>$P_2 := 0;$</td>
<td>return $R$</td>
<td>$L := t_l(T_1);$</td>
</tr>
<tr>
<td>$P_3 := H;$</td>
<td></td>
<td>$L := \text{getLeaf}(L, P_3);$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next()</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ($\neg \text{isLeaf}(P_3)$)</td>
</tr>
<tr>
<td>$P_3 := \text{hd}(P_3);$</td>
</tr>
<tr>
<td>else {</td>
</tr>
<tr>
<td>$P_3 := H;$</td>
</tr>
<tr>
<td>if ($P_2 = 0$)</td>
</tr>
<tr>
<td>$P_2 := 1;$</td>
</tr>
<tr>
<td>else {</td>
</tr>
<tr>
<td>$P_2 := 0;$</td>
</tr>
<tr>
<td>$T_1 := \text{nextSubtree}(P_1, T, H);$</td>
</tr>
<tr>
<td>$P_1 := \text{nextSubtree}(P_1, C, H);$</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
| }
```