CONS-Free Programs with Tree Input (extended abstract)

Amir M. Ben-Amram* Holger Petersen†

January 12, 1998

Abstract

We investigate programs operating on LISP-style input data that may not allocate additional storage. For programs without a test of pointer equality we obtain a strict hierarchy of sets accepted by deterministic, non-deterministic, and recursive programs. We show the classes accepted by programs with the ability to test pointers for equality are closely related to well-known complexity classes where relationships are mostly open.

1 Introduction

1.1 Background

Programming paradigms and related features of programming languages are subjects of continuing interest within the computer science community.

While all general purpose programming languages provide Turing completeness, there are obvious differences with respect to ease of use and efficiency. One of the few results establishing such discrepancies in a precise sense is the recent separation of pure and impure (destructive) LISP programs by Pippenger [7], where the two language variants are separated with respect to efficiency. In the present work we consider a more restrictive setting of sub-recursive languages. Their study is mainly motivated by the observation that for such restricted languages, it is meaningful to compare language variants with respect to *expressive power*, regardless of efficiency. This kind of study has led in the past to fundamental results that characterize the expressive power of various types of automata, and show that important complexity classes can be captured by restricted languages.

1.2 Outline

We start by introducing CONS-free programs, programs in an imperative language operating on a LISP-style input tree, but without access to dynamic ("heap") storage. The trees accepted by programs of this kind can be put into correspondence with strings over a finite alphabet, e.g. via a preorder encoding.

*The Academic College of Tel-Aviv, Tel Aviv, Israel, amirben@server.mta.ac.il

†Institute of Computer Science, Stuttgart University, Stuttgart, Germany, petersen@informatik.uni-stuttgart.de

This makes it possible to compare the power of CONS-free programs with that of resource-bounded Turing machines and related models that operate on string input. In the special case where the input is a linear list the correspondence to string languages is close and has been studied intensively, see [5, 4].

Our results investigate the case of general input. In this setting, we have been able to separate the acceptance power of deterministic CONS-free programs from those of more powerful programs, for example non-deterministic ones. This stands in contrast with the situation found for string languages, where the analogous questions are long-standing open problems.

Some of our results make use of techniques for simulating bounded counters, building on known relationships of counter machines to other automata.

A question arising in programming languages is the kind of access that the programmer has to memory pointers. In our framework the question appears in the form of whether CONS-free programs gain power from an equality test of pointers. We have been able to show that this capability does add power to our non-deterministic CONS-free language, while for other language variants the problem is still open.

2 Technical Background

2.1 Preliminaries

We consider programs written in a simple "flow-chart" language, i.e., an imperative language without block structure and recursion, and with a LISP view of data. Such programs have been considered by Jones [4, 5] and Pippenger [7].

The instructions available include `car`, `cdr`, and assignments. The former two operate on a CONS-cell and return the value of its respective field, which is either an atom or a pointer to another cell. Every instruction carries a label that the program may use as the destination of an unconditional or conditional jump, where the latter tests a variable for `nil`. Note that we have excluded `cons`; this makes our language weaker than LISP, for it cannot allocate storage. We call our language CF, an abbreviation of CONS-Free.

The unstructured form of programs is convenient in proving facts about the expressive power of our model. Often we will however present code segments using structured constructs like "`while`" and "`if...then...else`" that readily translate into the more restricted language.

In order to eliminate unnecessary details, we assume our data to consist solely of CONS-cells (to whom we also refer as *nodes*) and the single atom `nil`. Clearly it is possible to represent any finite set of atoms by choosing a suitable encoding such as `nil`, `(nil.nil)` etc. We denote the set of binary trees, all of whose leaves are `nil`, by $\mathbb{D}$. For a non-atomic tree T, we refer to its two subtrees as `hd` T and `tl` T (we thus distinguish between `hd` T and `car` T, which denotes a pointer to the root of the subtree).

The input to a program is a single element of $\mathbb{D}$, i.e., a single tree. We will concentrate on recognition problems and assume that the program has a way of signaling acceptance, say with special `accept` and `reject` instructions. We define however "rejection" as non-acceptance, so that a program that fails to terminate is considered as rejecting its input. The interesting question of whether termination can be ensured is mentioned briefly in a later section.

As a programming convention, we allow Boolean values `true` and `false` in our programs. These will be represented by some non-`nil` value and `nil` respectively. There is a caveat here, since this requires the input to be non-`nil`. We will tacitly assume that a test takes care of the special case of an input `nil` before entering the main body of the program. By using k variables of this kind, a finite counter with values in $[0, 2^k - 1]$ can also be maintained.

A natural extension, from a programming point of view, is procedure calling and recursion. The language so extended will be called RCF.

Extending our language in another direction we introduce non-determinism by a `choose` instruction that non-deterministically executes one of the operations supplied as its parameters. As usual we define acceptance by requiring that there be at least one successful sequence of choices. This variant of the language will be called NCF.

Since we consider CONS-free programs, that cannot allocate any new cells, program variables act as pointers into the input structure. The capability of testing whether two variables point to the same node will be introduced as another extension that can be added to each of the above languages, obtaining respectively the variants ECF, RECF, and NECF.

We give an example of a program that falls into the most restricted class CF of deterministic CONS-free programs that cannot test pointers for equality. Note that block structure is indicated by indentation.

```
E := true;
Y := X;
while Y do
  Y := cdr Y;
  if E then X := cdr X;
  E := not E
```

Considering the initial value of `X` as a LISP-style list, this program will move `X` to the middle of the list. This example shows an important programming technique whereby variables that point into the input tree can serve as counters, where here we have divided a counter by 2. The same technique has been used for counting by multihead automata. However, because the structure of the input tree can vary, more involved techniques will be necessary in order to simulate counters in our programs.

2.2 Classes of Sets Recognized by a Language

Since our languages are obviously sub-recursive, it is interesting to consider the class of sets of inputs that can be recognized by such a program. For a language L, We denote the class of sets that can be recognized by L-programs by a boldface **L**. We thus have the classes **CF**, **NCF** etc.

The fact that variables are constrained to point into the input structure brings to mind familiar computational models, in particular multihead automata. The classes of sets of strings (string languages) recognized by such devices are well studied. It is thus natural to compare our classes to the classes of string languages. However our classes comprise sets of trees, not of strings. This discrepancy can be settled in two ways. The first is to restrict the trees to linear lists that in the obvious manner encode strings. This way our languages coincide with familiar automata, and the classes of lists recognized are then well-known classes of string languages. This treatment can be found in Jones [5, 4].

In our work we have adopted a different approach: we translate between trees and strings using an encoding function $c_{\mathbb{D}} : \mathbb{D} \to \{0,1\}^*$. The particular function we use is based on a preorder listing of the tree and is defined by

$$\begin{array}{lcl} c_{\mathbb{D}}(\mathtt{nil}) & = & \mathtt{0} \\ c_{\mathbb{D}}(\mathtt{d}_1.\mathtt{d}_2) & = & \mathtt{1}c_{\mathbb{D}}(\mathtt{d}_1)c_{\mathbb{D}}(\mathtt{d}_2) \end{array}$$

Note that the set of strings that encode trees is obtained as the image set denoted by $c_{\mathbb{D}}(\mathbb{D}) \subset \{0,1\}^*$.

For any class $\mathcal{C}$ of subsets of $\{0,1\}^*$, we define the corresponding class of trees by

$$\text{tree-}\mathcal{C} = \{\, c_{\mathbb{D}}^{-1}(A) : A \in \mathcal{C} \,\}$$

Thus we consider, for instance, the class tree-LOGSPACE, which includes those sets of trees that, when transformed to strings, can be recognized by a logarithmic-space machine. For definitions of this class and other string classes such as NLOGSPACE and PTIME see [4]. The class of regular languages is denoted by REGULAR.

2.3 Counters as Storage Devices

In order to establish relations between our classes and string-language classes we will use simulations of *bounded counter machines*. We now briefly describe these machines and their relationship to Turing machines. Let $b(w)$ be a bound that depends on the input w. The counter machine has a read-only two-way input tape with one head and is equipped with a finite set of counters that can store a number in the range 0 to $b(w)-1$. A counter can be set to $b(w)-1$, decremented, and tested for zero. Note that an increment operation on a counter whose value is less than $b(w)-1$ can be simulated with the help of an auxiliary counter and a loop. Counter machines operate under the control of a finite program that may be deterministic, non-deterministic, or alternating. Taking a programming-language view, recursion can also be added (actually recursion has also been discussed in an automata-theory disguise of "machines with auxiliary pushdown store", see [2]).

The crucial fact that we exploit is the well-known correspondence of $b(w)$ bounded counters and Turing machines with space bound $\log b(w)$. The main example is the case of linearly bounded counters, $b(w) = \Theta(|w|)$, where deterministic, non-deterministic and alternating (or recursive) counter machines characterize the string-language classes LOGSPACE, NLOGSPACE, and PTIME, respectively. We refer to [4, 5] for a detailed discussion.

3 Programs without Test for Pointer Equality

We first summarize the basic inclusions between classes defined on trees and corresponding string representations.

Theorem 1 *The following relations hold:*

$$\begin{array}{rcl}\mathbf{CF} & \subseteq & \text{tree-LOGSPACE} \\ \mathbf{NCF} & \subseteq & \text{tree-NLOGSPACE} \\ \mathbf{RCF} & \subseteq & \text{tree-PTIME}\end{array}$$

Proof. The first two lines can be verified by a simple translation of CONS-free programs to Turing machines. The main points to note are that a pointer into the input can be represented in logarithmic space, and that logarithmic space suffices for the counter necessary for locating the `cdr` of a node in the string representation. For the third inclusion, the memoization technique of Cook is used [5]. □

We proceed to show that the first inclusion is strict.

Definition 2 *A node of a tree in* $\mathbb{D}$ *is called* even *if both of its children are* `nil`*, or both are non-*`nil`*. Otherwise the node is called* odd. EVEN *is the set of trees, all of whose internal nodes are even.*

EVEN has a very simple flavor, since it is defined by a local condition on each node; were the tree encoded as a string, a finite automaton would suffice for testing it.

Fact 3 $\text{EVEN} = c_{\mathbb{D}}^{-1}(\{0 + (1 + 100)^*\}) \in \text{tree-REGULAR}$.

In contrast, we show that this test is beyond the capabilities of CONS-free programs.

Theorem 4 *No* CF *program can recognize* EVEN. *This statement remains true, even if it is guaranteed that in the input there is at most one odd node.*

To prove the theorem, we have to introduce some definitions. The *configuration* of program p at a certain instant during its computation is $(l, X_1, \ldots, X_k)$ where l is the label of the instruction to be executed next, and $X_1, \ldots, X_k$ are the values of the k variables of the program. These values are either `nil` or references to nodes of the input tree T. The set of possible configurations is turned into the *configuration graph* $G_C(\mathrm{p}, T)$ by adding the usual successor relation $C_1 \to C_2$. A *computation path* of p on input T is a path in this graph that starts with the initial configuration $(l_1, T, \mathtt{nil}, \ldots, \mathtt{nil})$, where l_1 is the first instruction of the program, and ends in a terminal configuration (one that bears the label of a stop instruction). Since the program is deterministic, every (non terminal) configuration has exactly one successor.

For a variable X_i, by the *tree value* of X_i we mean the *tree* (element of $\mathbb{D}$) referenced by X_i — in contrast with the "pointer value" which is a particular node of the input tree. We call two configurations *equivalent* if they agree on the instruction label and on the *tree values* of all variables. A *situation* is an equivalence class of configurations under this relation. Since the program semantics only depends on tree values (recall that the language does not allow pointers to be compared), the *successor* of a non-terminal situation is well defined, and we obtain the *situation graph* $G_S(\mathrm{p}, T)$.

Lemma 5 *For a configuration C_1 the following diagram commutes, where a horizontal arrow represents the successor relation and vertical arrows map configurations to their equivalence class.*

$$\begin{array}{ccc} C_1 & \longrightarrow & C_2 \\ \downarrow & & \downarrow \\ [C_1] & \longrightarrow & [C_2] \end{array}$$

Proof. The lemma is proved by looking at each of the instructions in the language, and verifying that its action on equivalent configurations yields equivalent successors. □

Definition 6 *For $T \in \mathbb{D}$, $||T||$ is the number of non-isomorphic subtrees of T.*

Lemma 7 *Let* p *be a CF program, using k variables. The length of any terminating computation of* p *on input T is bounded by $|\mathrm{p}| \cdot ||T||^k$, where $|\mathrm{p}|$ is the number of instructions in* p.

Proof. A computation of length t corresponds to a path π of length t in the configuration graph. By Lemma 5 a corresponding path π', of the same length, exists in the situation graph. Because the program is deterministic (there is only one successor to a non-terminal configuration), the path π is simple, and we can use Lemma 5 to show that π' is simple too. Now, its length is bounded by the number of vertices in $G_S(\mathrm{p}, T)$, i.e., the number of situations, which is clearly bounded by $|\mathrm{p}| \cdot ||T||^k$. □

We say that program p *inspects* a node of the input tree, if at some point during its computation a variable refers to this node. Clearly, if a node is never inspected, the values of its children cannot affect the behavior of the program.

Proof (of Theorem 4). Consider a CF program p that allegedly recognizes EVEN. Let n be a number of the form $2^h - 1$ and let T_n be the perfectly balanced tree of height h (T_n has n nodes). Correctness of p implies that on input T_n, it terminates and accepts. Lemma 7 shows that the length of p's computation on T_n is bounded by $|p| \cdot h^k$, where p has k variables. Thus during its computation on T_n, the program inspects $O((\log n)^k)$ nodes. It follows that for all n larger than some constant, there are internal nodes of T_n (nodes whose children are not `nil`) which are never inspected. Let T'_n be a tree obtained by choosing one such node and changing one of its children to `nil`; p accepts T'_n, failing to recognize that it is not in EVEN. □

The second inclusion of Theorem 1 is also strict.

Theorem 8 *No* NCF *program can recognize the set* EVEN, *but its complement can be recognized.*

Proof. The proof of the negative result is a simple extension of the proof for the deterministic case. We omit the details.
For the positive result, here is the outline of a program to recognize $\mathbb{D} \setminus \mathrm{EVEN}$.
Set `X` to the root of the tree.
Non-deterministically follow a path in the tree, by setting `X` either to `car X` or to `cdr X`, until one of the children of `X` becomes `nil`.
If `X` also has a non-`nil` child, accept; else reject. □

Corollary 9 $\mathbf{CF} \subsetneq \mathbf{NCF} \neq co\text{-}\mathbf{NCF}$

This result contrasts with the situation for the related string-language classes, where separation of LOGSPACE from NLOGSPACE is a notorious open problem, while closure under complement holds for the non-deterministic class [8, Chapter 7].

As the last variation of the model without pointer equality we consider recursive programs. Recursion allows us (at last) to traverse a tree in preorder, and hence to simulate the action of a finite automaton on its preorder encoding; thus tree-REGULAR $\subset$ **RCF**. We present the procedure to simulate an automaton below. The procedure, `traverse`, calls auxiliary procedures `process0` and `process1` to simulate the action of the automaton when reading a 0 or a 1, respectively. The procedures will update the automaton's state, which they maintain as a finite counter. The result of the run (acceptance or rejection) will be determined from the state when the traversal is completed.

```
procedure traverse(T)
 if T ≠ nil then
   process1; traverse(car T); traverse(cdr T)
 else
   process0
```

An interesting extension of this technique allows for the simulation of Turing machines with $O(\log\log n)$ space.

Lemma 10 *For the input tree T of an RCF program let $t = ||T||$. Then the program can simulate a finite number of counters bounded by t.*

Proof. We use the lexicographic ordering of trees according to their binary preorder encoding. This ordering can be checked by a recursive function `lexgt?` that returns `true` if its first parameter is strictly greater than its second. Let the decreasing sequence of subtrees of T be $t_{t-1} = T, \ldots, t_0 = \texttt{nil}$. A counter value i can be maintained by having a variable pointing to t_i. A counter is zero if and only if the corresponding variable is `nil`. Decrement can be implemented by finding the lexicographically preceding subtree (procedure `lexpred`). Setting a counter to the maximum value of $t-1$ is achieved with a recursive procedure `lexmax`.

We give a detailed description of `lexpred` only. It accesses two global variables `X` and `Y`, where `X` points to the old tree and `Y` receives the new one. Initially `Y` $=$ `nil` and `lexpred` is called with the input tree.

```
procedure lexpred(Z)
 if lexgt?(X, Z) and lexgt?(Z, Y) then Y := Z;
 if Z ≠ nil then
   lexpred(car Z); lexpred(cdr Z)
```

□

Lemma 11 *For an input tree T with n nodes we have $||T|| > \lceil \log n \rceil$.*

Proof. A binary tree with $n > 0$ nodes (CONS-cells) contains a path from its root to one of its leaves of length at least $\lceil \log_2(n+1) \rceil$. Thus there are $\lceil \log_2(n+1) \rceil + 1$ non-isomorphic subtrees in T rooted at the nodes of this path. □

We now show how the counters described in Lemma 10 can be used to simulate Turing machines.

Note: in the definition of acceptance by $O(\log \log n)$-space machines, we allow the weak mode of acceptance, where the space bound has to be obeyed only on accepted inputs. For a discussion of weak versus strong modes of space-bounded acceptance, see [8]; we will contend ourselves with citing the fact that the weak class strictly contains the strong class.

Theorem 12 tree-DSPACE$(\log \log n) \subsetneq$ **RCF**.

Proof. First note that the inclusion must be strict, due to the deterministic space hierarchy (consider the restriction to lists). We proceed to show that the inclusion holds.

Let the number of nodes of T be n; then the length of its string representation is $2n+1$. According to the Lemma 10 we can maintain in the RCF programs a finite number of global variables that represent $||T||$ bounded counters. These counters can represent $O(\log ||T||)$ bits of the Turing machine work tape. Since (by the last lemma) $||T|| > \lceil \log n \rceil$, we can simulate any Turing machine whose space complexity is $O(\log \log n)$.

We enhance the `traverse` procedure shown above to simulate a bidirectional input head. We now assume that the `process` procedures—in addition to updating the representation of the machine's internal state and working storage—return the value "true" if the Turing machine's input head is to be moved to the right (forwards), and "false" if it is to move left (backwards). Similarly, we change `traverse` to accept a second parameter, `Dir`, indicating the direction in which traversal is to proceed.

Since Turing machines with read-only input tape are supposed to use endmarkers, we add procedures `process2` and `process3` to process the left and right endmarker, respectively. The simulation will be initiated by procedure `simulate`, which calls the recursive procedure `traverse2way` (Figure 1). We assume that the initial position of the input head is on the left endmarker, and that moving off an endmarker terminates the program.

A further technical remark: we implement the operation of returning a value from a procedure by assigning it to a global variable that is reserved for this purpose. Therefore it is permissible for a function to return without explicitly defining a return value; the last return value rests unchanged. □

The simulation given in the preceding proof can be enhanced as follows. In an initial phase, when the input is not yet being traversed, it is possible to simulate non-deterministic and even alternating operation of the Turing machine by means of recursion (essentially changing non-deterministic search into deterministic backtracking). This extension uses the fact that our simulator can be forced to stop on every input (hence an unfortunate guess cannot cause it to run wild). We can thus simulate machines with "blind non-determinism" as well as "blind alternation" [6, 3, 1].

One consequence of the simulation is that Freivalds' result, that $\{a^p b^q \mid p, q \geq 0, p \neq q\} \in$ DSPACE$(\log \log n)$ [8, Lemma 4.1.3], can be adapted to show

```
procedure simulate(T)
  1:  if process2 then goto 2 else return
  2:  if traverse2way(T, true)  then goto 3 else goto 1
  3:  if process3 then return else goto 4
  4:  if traverse2way(T, false) then goto 3 else goto 1

procedure traverse2way(T,Dir)
  if T=nil then
   process0
  else
  if Dir then goto 1 else goto 5
  1:  if process1 then goto 2 else return
  2:  if traverse2way(car T, true)  then goto 3 else goto 1
  3:  if traverse2way(cdr T, true)  then return else goto 4
  4:  if traverse2way(car T, false) then goto 3 else goto 1
  5:  if traverse2way(cdr T, false) then return else goto 4
```

Figure 1

that the following set belongs to **RCF**: BAL $= \{T \in \mathbb{D} : |\text{hd } T| = |\text{tl } T|\}$. This is interesting because the problem seems at first sight difficult to solve in the CONS-free language.

The counting technique of Lemma 10 can also be used to show the relationship between the classes **NCF** and **RCF**.

Theorem 13 **NCF** $\subsetneq$ **RCF**.

Proof. The inclusion follows by simulating non-determinism with recursion, as mentioned above. The classes are separated by the language EVEN. □

4 The Pointer Equality Test

Inspecting the proof of Theorem 4 reveals that it ceases to apply if the language is extended by the capability to test whether two pointers refer to the same node. Thus it seems that the equality test, which we denote by eq, may affect the expressive power of the language (this is another feature which does not appear if the input is restricted to linear lists, where the test can be simulated with the help of an auxiliary variable and one loop). Recall that the basic language with equality test is named ECF, with the corresponding extensions NECF and RECF.

Theorem 14 *The following relations hold:*

$$\begin{aligned} \textbf{ECF} &\subseteq \text{tree-LOGSPACE} \\ \textbf{NECF} &\subseteq \text{tree-NLOGSPACE} \\ \textbf{RECF} &\subseteq \text{tree-PTIME} \end{aligned}$$

Proof. Analogous to the proof of Theorem 1. □

While we obtained a strict hierarchy for the classes of sets accepted by programs without a test for pointer equality, the situation is much more complex for the classes admitting this operation.

We have not been able to separate the class **ECF** from either **CF** or **NECF**. A separation from the latter class would probably require an argument exploiting the fact that the input is a tree. If the classes could be separated on linear input structures this would imply that LOGSPACE and NLOGSPACE can be separated, since on linear lists the inclusions of Theorem 14 are equalities.

In contrast with the case of **ECF**, for non-deterministic and recursive programs we have been able to extend the inclusions to equalities for general input.

Theorem 15 **NECF**= tree-NLOGSPACE.

Proof. As pointed out in Section 2.3 it suffices to show that an NECF program can simulate a non-deterministic counter program, whose counters have integer values bounded by a constant times the input size. Our construction achieves this with counters bounded by half the input size.

Let p be a non-deterministic $\lfloor |w|/2 \rfloor$-bounded counter program that operates on the string $c_{\mathbb{D}}(T)$. We translate it to an NECF program that operates on T by representing every counter as a pointer into the input tree, where the value of the counter is given by the preorder number of the node (note that this is a different method than used for programs without pointer comparison). In particular, the root of the tree represents the value zero. If T has n nodes, $|c_{\mathbb{D}}(T)| = 2n + 1$, so the desired counter size will be achieved.

Using this representation, testing a counter for zero is trivial, setting to $n-1$ is quite simple, so we will describe in more detail only the decrement operation.

Let ν be a node of the tree, and π its parent. We use the following facts:

1. If $\nu = \texttt{car}\,\pi$, the predecessor of ν in preorder is π.

2. If $\nu = \texttt{cdr}\,\pi$, the predecessor is the lowest node reachable by pursuing `cdr` pointers, starting at $\texttt{car}\,\pi$.

This gives the following procedure. Here non-determinism is used. The operation `choose` is the non-deterministic choice. The statement `fail` indicates that no answer is reached (this could be expressed by looping forever). The counter which we aim to decrement is `Y`; `Z` is a temporary variable. We assume that `Y` does not point to the root (which has preorder number zero).

```
Z := root    (* search for the parent of Y *)
while (cdr Z ≠ Y) and (car Z ≠ Y) and (Z ≠ nil)
  Z := choose(car,cdr) Z;
if Z = nil then
  fail
else
  if Y = car Z then    (* case 1 *)
    Y := Z
  else                 (* case 2 *)
    Y := car Z;
    while cdr Y ≠ nil
      Y := cdr Y
```

The input head is represented in much the same way as a pointer into the tree which can be moved forwards or backwards using a similar procedure. This completes the description of the translation and hence the proof of the theorem. □

Theorem 16 **RECF**= tree-PTIME.

Proof. It is known (see for example [5]) that adding recursion to linearly-bounded counter programs gives a model that captures PTIME. Our theorem thus follows by showing how to compile linearly-bounded counter programs into RECF. The two languages have the same control structures: procedures, sequential execution and branching, so we only have to replace the instructions that manipulate the counters. As before, the involved operation is decrement, where we now use recursive traversal in order to find the predecessor of a node. Procedure `pred` below sets variable `Y` to the predecessor of `X`, which is again assumed to be different from the root. During the traversal the Boolean variable `F` (initially true) records whether the node references by `X` has already been discovered. The procedure is called with the input tree.

```
procedure pred(Z)
 if Z ≠ nil then
   if Z = X then F := false;
   if F then Y := Z;
   pred(car Z); pred(cdr Z)
```

□

5 Some Recent Results

In this abstract we have considered the classes of trees accepted by CONS-free programs with or without the pointer equality test, non-determinism and recursion. An extension that could not be included is alternating programs. It can be shown that for both our language variants, CF and ECF, alternation gives the same expressive power as recursion.

Further results about closure properties have been obtained. The classes **NECF** and **NRCF** are known to be closed under complement because of Theorems 15 and 16. As known for Turing machines, closure under complement is related to the possibility of guaranteeing termination of the programs. Using various constructions, this has been shown possible for the classes **ECF**, **RCF** and **CF** (note that the question is non-trivial precisely because these classes do not seem to coincide with related complexity classes).

6 Conclusion

CONS-free programs, with input restricted to linear lists, can be equated with "conventional" types of automata, with the result that their expressive power can be nicely characterized. With general tree input, the situation is much different, and so far largely unexplored. In this work we have been able to identify many of the relations between classes accepted by such programs with different language features and compare them to related classes of string languages.

Many questions remain open, such as the relationship of **ECF** to LOGSPACE and of **RCF** to PTIME. These relationships have to do with the interesting question of whether pointer comparison increases the power in the respective languages. Another interesting direction is to consider non-tree inputs, for example by allowing shared CONS-cells.

References

[1] G. Buntrock, F. Drewes, C. Lautemann, and T. Mossakowski. Some modifications of auxiliary pushdown automata. *Informatique théorique et Applications/Theoretical Informatics and Applications*, 25:545–556, 1991.

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

[3] B. Jenner and B. Kirsig. *Alternierung und logarithmischer Platz.* PhD thesis, Universität Hamburg, 1989.

[4] N. D. Jones. *Computability and Complexity — From a Programming Perspective.* MIT Press, Cambridge, Mass., London, England, 1997.

[5] N. D. Jones. LOGSPACE and PTIME characterized by programming languages. 1997. Submitted.

[6] I. Niepel. *Logarithmisch-platzbeschränkte Komplexitätsklassen—Charakterisierung und offene Fragen.* Diplomarbeit, Fachbereich Informatik der Universität Hamburg, 1987.

[7] N. Pippenger. Pure versus impure Lisp. *ACM Transactions on Programming Languages and Systems*, 19:223–238, 1997.

[8] A. Szepietowski. *Turing Machines with Sublogarithmic Space.* Number 843 in Lecture Notes in Computer Science. Springer, Berlin-Heidelberg-New York, 1994.