A Precise Version of a Time Hierarchy Theorem[*]

Amir M. Ben-Amram
The Academic College of Tel-Aviv Yaffo
Tel-Aviv, Israel
`amirben@server.mta.ac.il`

Neil D. Jones
Neil D. Jones[†]
Department of Computer Science (DIKU)
The University of Copenhagen, Denmark
`neil@diku.dk`

Abstract. Using a simple programming language as a computational model, Neil Jones has shown that a constant-factor time hierarchy exists: thus a constant-factor difference in time makes a difference in problem-solving power, unlike the situation with Turing machines. In this note we review this result and fill in the programming details in the proof, thus obtaining a *precise version* of the theorem with explicit constant factors. Specifically, multiplying the running time by 232 provably allows more decision problems to be solved.

Keywords: time complexity, hierarchy theorem, universal program.

1. Introduction

Many textbooks in Theoretical Computer Science include the *Turing machine speedup theorem*, stating that any desired constant factor can (almost) always be saved in the running time of a

[*]This research was partially supported by the Danish *DART* project.
[†]Address for correspondance: Department of Computer Science (DIKU), The University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark.

Turing machine. Hence constant factors do not matter in time complexity. In [3], Neil Jones claimed that in real-world programming constant factors *do* matter, and hence a theorem which demonstrates this situation in a formal model may be more relevant than the Turing machine result. A related point is that in practice algorithms are most often implemented in software rather than hardware, the programmer not being able to accelerate the hardware at will. Jones proposed to use as a computational model a simple programming language rather than a class of machines. For a simple imperative language, called `I` and operating on LISP-style data, he proved a hierarchy theorem showing that a constant-factor increase in time allows more decision problems to be solved. The proof has the customary structure of proofs by diagonalization, using a universal program (also called a self-interpreter).

The purpose of this work is to demonstrate that programming in Jones' language is convenient enough to write the programs involved in the proofs, thus explicitly and without too much effort obtaining a precise proof where constant factors are explicitly calculated based on programs. We thus obtain versions of the hierarchy theorems with explicit constant factors and moreover establish that these constants are not very large.

Instead of the original presentation from 1993 we base this work on the expanded presentation from 1998 [1]. For completeness of this paper, we give the necessary background in the next two sections. Further discussion of the result and of related work will be given at the conclusion.

2. Background

Language `I` is an imperative language whose single data type is the type of complete ordered binary trees. *Complete* means that every internal node has two children, which we call the head and tail. The leaves are called *atoms*. We write trees using LISP syntax as dotted pairs `(a.b)` or as lists `(a b c)` = `(a.(b.(c.nil)))`. In `I` there is only a single atom, `nil`. The set of trees over this single atom we denote by $\mathbb{D}$. The *size* of $\mathtt{d} \in \mathbb{D}$ is defined inductively: $|\mathtt{nil}| = 1$, and $|\mathtt{d}_1 . \mathtt{d}_2| = 1 + |\mathtt{d}_1| + |\mathtt{d}_2|$.

Operators for manipulating trees are the constructor `cons` and the destructors `hd` and `tl`. An `if` command tests whether the conditional expression evaluates to a non-`nil` value (signifying "truth") or to `nil` ("falsehood"). A `while` command executes until the condition becomes `nil`.

The complete syntax of the language is shown in Table 1 together with the *concrete syntax* according to which every program is presented as a tree. Note that a program only has a single variable. The concrete syntax makes use of extra atoms besides `nil`: "`while`", "`:=`" etc. To reduce our data type to $\mathbb{D}$ we select for each special atom a code from $\mathbb{D}$. These atoms should thus be considered a mere notational convenience. Similarly it may be convenient to use more than a single variable. Writing a program in this way we then have to translate it into `I` by packing all our variables into the single variable `X`. The way of packing can be also arbitrary; when we get to actual programming we will be more specific.

Running time for `I` programs is defined by a unit-cost model that reflects the cost associated with customary LISP implementations. Thus accessing a constant or a variable, applying an operator, testing or copying the value of an expression each cost one time unit. The partial

Syntactic category		*Informal syntax*	*Concrete syntax*
P : Program	::=	`read X; C; write X`	`C`
C : Command	::=	`X := E`	`(:= E)`
	|	`C1; C2`	`(; C1 C2)`
	|	`if E then C1 else C2`	`(if E C1 C2)`
	|	`while E do C`	`(while E C)`
E : Expression	::=	`X`	`x`
	|	`D`	`(quote D)`
	|	`hd E`	`(hd E)`
	|	`tl E`	`(tl E)`
	|	`cons E1 E2`	`(cons E1 E2)`
D : Data Value	$\in$	$I\!\!D$	

Table 1: Syntax of language I*: informal and concrete*

function computed by program p is denoted by $[\![\mathtt{p}]\!]$. The running time of the program on input d is denoted by $time_{\mathtt{p}}(\mathtt{d})$ (∞ or a natural number).

The time hierarchy theorems for I are proved by a diagonalization argument which makes use of a *timed self-interpreter*, an I program to execute I programs under a time limit. We use the following representation for integers in our language: integer n is represented as $\mathtt{nil}^n$, a list of n nil's.

Definition 2.1. I-program tu is an *efficient timed self-interpreter* for I if there are constants k and l such that for every I program p, every $\mathtt{d} \in I\!\!D$ and $n \geq 1$:

If $time_{\mathtt{p}}(\mathtt{d}) \leq n$ then $[\![\mathtt{tu}]\!](\mathtt{p}\,.\,\mathtt{d}\,.\,\mathtt{nil}^n) = ([\![\mathtt{p}]\!]\,\mathtt{d}\,.\,\mathtt{nil})$

If $time_{\mathtt{p}}(\mathtt{d}) > n$ then $[\![\mathtt{tu}]\!](\mathtt{p}\,.\,\mathtt{d}\,.\,\mathtt{nil}^n) = \mathtt{nil}$

And in both cases $time_{\mathtt{tu}}(\mathtt{p}\,.\,\mathtt{d}\,.\,\mathtt{nil}^n) \leq k \cdot min(n, time_{\mathtt{p}}(\mathtt{d})) + l \cdot |\mathtt{p}|$.

Note that the constants k, l are quantified *before* the program p. This program-independence of the slowdown factors is vital to our result.

We now describe the structure of an efficient self-interpreter program, tu, for I. First we give its "outline," omitting a central fragment which we call STEP. This fragment performs a single interpretation step using three state variables: Cd is used as a stack to keep code that should be executed. Stk is a value stack for evaluating nested expressions. X contains the current contents of the single variable of the program. In addition we have a counter variable Cntr to hold the time remaining. The program appears in Figure 2; we remark that the connective *and* is not in the language. Consider it to be syntatic sugar. As for STEP, in [1] we refrained from giving program code, and found it more informative to present the program as a collection of rewriting rules. Blank entries in Table 3 correspond to values that are neither referenced nor changed in a rule; a "+" in the last column marks rules whose execution is followed by decreasing Cntr by one.

```
read X;              (* X = (p.d.nil^n) *)
Cd := cons (hd X) nil;
Cntr := tl tl X;
Stk := nil;
X := hd tl X;
while Cd and Cntr do STEP;
if Cd then
  X := nil
else
  X := cons X nil;
write X
```

Figure 2: Program tu.

3. Hierarchy Theorems

The following hierarchy theorem is from [1]. Here $\mathrm{TIME}^{\mathtt{I}}(T(n))$ is the class of subsets of $I\!D$ that are decidable by I programs in time bounded by $T(n)$ for input of size n.

Theorem 3.1. *For every time-constructible function $T(n) \geq n$, there is a constant b such that there are sets in* $\mathrm{TIME}^{\mathtt{I}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}}(T(n))$.

A specialized version for linear time runs as follows, where $\mathrm{TIME}^{\mathtt{I}}(an)$ is the class of sets that can be decided by I programs in time bounded by an.

Theorem 3.2. *There is a constant b such that for all integer $a \geq 1$,* $\mathrm{TIME}^{\mathtt{I}}(abn) \setminus \mathrm{TIME}^{\mathtt{I}}(an)$ *is not empty.*

The proofs of the two propositions are similar; we repeat the proof for the linear-time case. It hinges on a typical diagonalization argument, using the following program diag:

```
read X;
Timebound := nil^{a·|X|};
X := tu(X,X,Timebound);
if hd X then X := nil else X := (nil.nil);
write X
```

The first assignment is a shorthand for a piece of code that performs the desired computation, i.e., computing the size of a tree and representing it as a list of nil's. The computation of $\mathtt{nil}^{a\cdot|\mathtt{X}|}$ can be performed in time bounded by $c \cdot a \cdot |\mathtt{X}|$ for some constant c. Next comes the invocation of tu, whose running time on program p with time bound n can be bounded by $k \cdot n + l \cdot |\mathtt{p}|$ for constants k and l. On input $\mathtt{d} = \mathtt{p}$, program diag runs in time at most $e + (c + k) \cdot a \cdot |\mathtt{p}| + l \cdot |\mathtt{p}|$ where e accounts for the time beyond computing Timebound and running tu.

Cd	Stk	X	⇒	Cd	Stk	X	+
(; C1 C2).Cd			⇒	C1.C2.Cd			
(:= E).Cd			⇒	E.do_assgn.Cd			
do_assgn.Cd	w.s	v	⇒	Cd	s	w	+
(if E C1 C2).Cd			⇒	E.do_if.C1.C2.Cd			
do_if.C1.C2.Cd	(t.u).s		⇒	C1.Cd	s		+
do_if.C1.C2.Cd	nil.s		⇒	C2.Cd	s		+
(while E C).Cd			⇒	E.do_while. (while E C).Cd			
do_while. (while E C).Cd	(t.u).s		⇒	C.(while E C).Cd	s		+
do_while.C1.Cd	nil.s		⇒	Cd	s		+
x.Cd	s	v	⇒	Cd	v.s	v	+
(quote D).Cd	s		⇒	Cd	D.s		+
(hd E).Cd			⇒	E.do_hd.Cd			
do_hd.Cd	(t.u).s		⇒	Cd	t.s		+
do_hd.Cd	nil.s		⇒	Cd	nil.s		+
(tl E).Cd			⇒	E.do_tl.Cd			
do_tl.Cd	(t.u).s		⇒	Cd	u.s		+
do_tl.Cd	nil.s		⇒	Cd	nil.s		+
(cons E1 E2).Cd			⇒	E1.E2.do_cons.Cd			
do_cons.Cd	u.t.s		⇒	Cd	(t.u).s		+
nil			⇒	nil			

Table 3: The STEP *macro, expressed by rewriting rules*

By $Acc(\texttt{diag})$ we denote the set of trees accepted by diag, that is, the set of $\texttt{d} \in \mathbb{D}$ for which $[\![\texttt{diag}]\!]\,\texttt{d} \neq \texttt{nil}$. Letting $b = c + k + l + e$, we have $Acc(\texttt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(abn)$.

Now suppose for the sake of contradiction that $Acc(\texttt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(an)$. Then there exists a program p with $Acc(\texttt{p}) = Acc(\texttt{diag})$, and $time_{\texttt{p}}(\texttt{d}) \le a \cdot |\texttt{d}|$ for all $\texttt{d} \in \mathbb{D}$. Consider the application of p to its own code. The time bound on p implies that $[\![\mathrm{tu}]\!](\mathrm{p.p.nil}^{a|\texttt{p}|}) = ([\![\mathrm{p}]\!]\,\texttt{p . nil})$. If $[\![\mathrm{p}]\!]\,\texttt{p}$ is nil, then $[\![\texttt{diag}]\!]\,\texttt{p} = \texttt{(nil.nil)}$. If it is not nil, then $[\![\texttt{diag}]\!]\,\texttt{p} = \texttt{nil}$. Both cases contradict $Acc(\texttt{p}) = Acc(\texttt{diag})$, so the assumption that $Acc(\texttt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(an)$ must be false.

Theorem 3.1 is proved by replacing the computation of Timebound with code for computing $\texttt{nil}^{T(|\texttt{x}|)}$. The explicit constructions described in this paper allow us to replace the constants c, k, l, e in the proof with explicit values. This way we obtain the following *precise versions* of the hierarchy theorems:

Proposition 3.1. *For all* $a \ge 1$, $\mathrm{TIME}^{\mathtt{I}}((126a + 106) \cdot n) \setminus \mathrm{TIME}^{\mathtt{I}}(an)$ *is non-empty.*

Proposition 3.2. *Let $T(n)$ be a time-constructible function such that the program constructing $T(n)$ runs in time bounded by $aT(n)$.*
Then $\mathrm{TIME}^{\mathtt{I}}((a+124)\cdot T(n)+108n)\setminus \mathrm{TIME}^{\mathtt{I}}(T(n))$ *is non-empty.*

4. Implementation and Analysis of STEP

In order to evaluate the running time of non-trivial program segments, we have coded them in full, tested them for correctness and analyzed their running time using auxiliary programs. It is easy to realize that the diagonalization proof will give a tighter result with a more efficient timed interpreter. Therefore, we have attempted to code it carefully and in a time-efficient manner.

The running time of a single application of STEP depends on the rule chosen, and is the sum of the time spent on choosing the rule and the time of executing the rule itself: we refer to these quantities as *branching time* and *rule time*. The worst-case time of STEP is the maximum sum of branching time and rule time over the different rules.

4.1. Encoding of Multiple Variables and Atoms

In order to code the program in the `I` language we have to decide on a representation of the four variables and the many special atoms in terms of one single variable `X` and the single atom `nil`.

We choose to represent the variables of STEP by setting `X` to

`(Cd.(Stk.(Cntr.Val)))`

Thus `hd Cd` is accessed by `hd hd X`, etc. The special atoms that have to be encoded are the following[1]:

```
; := do_assgn if do_if while do_while X quote hd do_hd tl do_tl
cons do_cons
```

The atoms can be divided in two groups. One group includes atoms that appear at the start of a list: these are the atoms that are part of the `I` concrete syntax, for example `while` appears in the list `(while E C)`. The other atoms appear only on their own: these are the atoms used internally by the step macro, for example `do_while`.

The first distinction that the top-level of STEP has to do is between these two cases. We have made it easy to distinguish between the two groups of atoms by encoding all stand-alone atoms as values whose head is `nil`. Since all special atoms are encoded as non-`nil` values, it is now possible to decide which case is before us by looking at `hd hd Cd`. Except for that, the particular encoding is quite arbitrary, and was chosen so that rules that take longer time to execute will be faster to recognize, thus minimizing the worst-case time of STEP.

[1]We processed our `I` programs using the CMU Common LISP system. Because of Common Lisp's restrictions we used the atom `$` used instead of the atom `;` in some of the program texts.

`do_if`	`(nil nil)`
`do_cons`	`(nil nil nil)`
`do_while`	`(nil (nil))`
`do_tl`	`(nil (nil) nil)`
`do_hd`	`(nil (nil nil))`
`X`	`(nil (nil nil) nil)`
`do_assgn`	`(nil ((nil) nil) nil)`
`if`	`(nil)`
`quote`	`(nil nil)`
`:=`	`((nil))`
`;`	`((nil) nil)`
`cons`	`(((nil)))`
`while`	`(((nil)) nil)`
`tl`	`(((nil) nil))`
`hd`	`(((nil) nil) nil)`

Table 4: Encoding for atoms, written in list notation.

4.2. Implementation and Analysis

Given the encodings, writing I code for the STEP macro is straight forward. This code appears in file `step.i` (Appendix 1). A *time analysis program* was written (in LISP) to compute the running time of individual rules and of the whole macro. The program operates according to a simple recursive formula for the time complexity of an expression. The output from this program follows; it should be self-explanatory.

```
(('HD 48) ('TL 48) ('WHILE 48) ('CONS 55) ('$ 46) (':= 52)
('QUOTE 44) ('IF 49) ('DO_ASSGN 54) ('X 58) ('DO_HD 53)
('DO_TL 53) ("DO_WHILE:t" 59) ("DO_WHILE:f" 53) ('DO_CONS 52)
("DO_IF:t" 52) ("DO_IF:f" 53) ('NIL 14))
```

These figures include the branching time. The worst-case time of STEP is thus 59.

5. Running Time of the Interpretation Loop

The main loop in program `diag` is the interpretation loop, at the heart of the timed interpreter `tu`. This loop was given in Section 2 as follows:

```
while Cd and Cntr do STEP
```

For the sake of conciseness, we change this to

```
while Cntr do STEP
```

Macro STEP has a rule for the case of empty Code stack, in which we set `Cntr` to `nil`; this preserves the meaning of the loop, except for the limit case in which `Cd` becomes empty at the very step where `Cntr` goes down to zero. In this case our program will conclude incorrectly that the interpreted program has outrun the time limit. Thus, the condition $time_{\mathtt{p}}(\mathtt{d}) \leq n$ in the specification of `tu` will change to $time_{\mathtt{p}}(\mathtt{d}) < n$. But this can be easily taken care of by making `Cntr` larger by one at the start.

Let $T_{\mathrm{STEP}}(n)$ be the time spent inside the interpretation loop (i.e., not counting loop control). We now bound $T_{\mathrm{STEP}}(n)$. We divide the iterations of `STEP` in two groups, the marked rules and the unmarked rules, where the marked rules are those which also decrease `Cntr`. If N_1 is the number of applications of marked rules and N_2 that of unmarked rules, we have $N_1 \leq n$. To bound N_2 we consider the code stack `Cd`. We note that only marked rules pop `Cd`. It follows that we can bound N_2 in terms of N_1 and R, the number of values left at the stack when `Cntr` goes to zero. Some marked rules (ending in `+`) pop two stack elements; thus to compare N_2 with N_1 it is important to observe that every rule that pops two values is matched by a preceding rule that pushes at least two. Given this observation, we have $N_2 \leq N_1 + R$. If we let t_m stand for the maximal running time of a marked rule, and t_u for that of an unmarked rule, we obtain $T_{\mathrm{STEP}}(n) \leq nt_m + (n+R)t_u$.

The fact that we increased the value of `Cntr` by one does not have to be taken into account — the reader may verify that our bound is loose enough to absorb this.

We now bound the value of R. The code stack initially contains the code of the program `p`, and the elements later pushed onto it are fragments of `p`. We observe that before pushing the stack, the program always verifies that `hd hd Cd` is non-`nil`; and the fragment that is pushed is taken from `tl hd Cd`. This implies that the size of the fragments decreases by at least 4 at a time as we go up the stack, and we obtain $R \leq \frac{1}{4}|\mathtt{p}|$. The values of t_m and t_u are 59 and 55 respectively (see the list at the end of the last section), so

$$T_{\mathrm{STEP}}(n) \leq 114n + \frac{55}{4}|\mathtt{p}| \leq 114n + 14|\mathtt{p}|\,. \qquad (1)$$

6. Computing `Timebound`

The computation of `Timebound` is made using an `I` program for computing the size of a tree, essentially from [1]. Complete `I` code appears in file `tree-size.i` (Appendix 2), which was actually run and tested under Common Lisp. The program starts with the input tree `d` in the variable `X` and ends with `X=(nil.|d|.d)` so that both the tree and the size are available for further processing.

Running time: Consider the two branches inside the `while` loop. The first branch does not increment the size, but moves an internal node of the tree into the rightmost path. No node is ever moved out of the rightmost path so the number of such iterations is bounded by the number of internal nodes less one, i.e., $(|\mathtt{d}|-1)/2 - 1 = (|\mathtt{d}|-3)/2$.

```
read X;
Compute Timebound;
X := (cons (cons (tl tl X) nil)
     (cons nil (cons (cons nil (tl X)) (tl tl X))));
while (hd tl tl X) do STEP;
if (hd X) then       (* not finished on time *)
  X := nil
else      (* finished on time, invert result *)
  if (tl tl tl X) then
    X := nil
  else
    X := X;
write X
```

Figure 5: Program diag.

The second branch increments the size by two, while reducing the tree, so it is clearly executed $(|\mathtt{d}| - 1)/2$ times.

We obtained the cost of each of the branches, as well as the non-iterative part, by applying the time analysis program. The results appear as annotations in the enclosed source code. An easy calculation (omitted here) gives the final result, which in terms of $n = |\mathtt{d}|$ amounts to $23.5 \cdot n - 38.5$. Taking into account that in our application $n \geq 5$, we round this up to $24n - 41$.

So far our program computed $|\mathtt{d}|$. Program diag calls for computing $a \cdot |\mathtt{d}|$. To obtain this result we just have to change the initial value of the counter from 1 to a and increment the counter by $2a$ instead of 2 each time. Adapting our calculations to this change gives the following result. We denote the time for this part by $t_{\mathrm{bound}}(n, a)$:

$$t_{\mathrm{bound}}(n, a) \leq (22 + 2a)n - 2a - 39 \,. \tag{2}$$

7. Completion of the Program

We now complete program diag, as used for the linear time hierarchy. In contrast with the presentation in Section 2, here we do not write tu separately; instead we write program diag directly, using the pieces we already have. The resulting program appears in Figure 5. The assignment after computation of Timebound takes the contents of X, that include the time bound and the original input (see last section), and creates the structure that STEP expects (Cd.Stk.Cntr.Val).

Running time. Let n denote the size of the input. The time for computing Timebound is given by (2). The non-iterative parts before and after the main loop take at most 27 time units.

Consider the main loop: the time spent in all the iterations of STEP put together is given by substituting an for the time bound in Eq. (1) to obtain $114an + 14n$. From the reasoning leading to that bound, we can also deduce that the *number* of iterations is bounded by $2an+14n$. Add one for the execution of the `while` command that fails, multiply by the time of the `while` command itself (5 units), and we obtain the expression $10an + 70n + 5$. Putting the pieces together we obtain:

$$(22 + 2a)n - 2a - 39 + 27 + 114an + 14n + 10an + 70n + 5 < (126a + 106)n$$

We conclude that $Acc(\texttt{diag}) \in \mathrm{TIME}^{\mathrm{I}}((126a+106)n)$. The diagonal argument shows that it cannot be in $\mathrm{TIME}^{\mathrm{I}}(an)$, proving Proposition 3.1. For the general case of a time-constructible bound $T(n)$, we use the program we gave to compute the input size (as for $a = 1$), followed the program that computes $T(n)$. Adjusting the calculations gives Proposition 3.2.

8. Discussion

In this work we have partially answered two questions regarding the language I. The first is: how efficient a self-interpreter for this language can be. The other is: for what constants can we prove the constant-factor hierarchy theorem. The questions are related in that we prove the hierarchy theorem by means of a self-interpreter. The two questions are not completely resolved: we have no evidence to the optimality of our self-interpreter, except that we do not see a way to make it significantly faster. Further, it is perfectly possible that another proof of the hierarchy theorem exists besides the diagonalization proof. Such a proof could possibly provide smaller constants.

In fact we do not even know whether there is a smallest constant for the hierarchy theorem. With appropriate formulation it could be that the hierarchy is dense, in the sense that multiplying the coefficient in the time bound by any constant larger than one could buy more decision power. The precise nature of the time hierarchy for a given computational model depends clearly very much on the specifics of the model. The more realistic is the computational model, the more can we consider the result to be saying something about "reality". We see this work as an indication of the kind of results that can be achieved with a fairly realistic model.

Our results can be related to two kinds of previously published work. First, constant-factor tight hierarchy theorems have been proved for certain machine models, including variants of random access machines and Turing machines [7, 9]. However these works had either used very artificial models or did not include full program code and computation of the constant factor. Another kind of related work is attempts to produce a "minimal" universal program [2, 4, 5, 6, 8]. However the criterion that these authors tried to optimize was size rather than speed, motivated by questions like "what is the smallest machine that is, in a certain sense, universal". To this end they allow much flexibility in choosing the encoding for the input machine. In contrast, our self-interpreter uses a concrete syntax which is "natural" in the sense that it resembles the concrete syntax of LISP. Indeed, all the previous work we mentioned related to computation with abstract machines rather than with a naturally-looking programming language. This difference in way of thinking may have pushed us to attempt different kind of results. It is well known that working with a different framework causes different questions to be asked. Our conviction is that the shift to programming-oriented models is desirable in that the questions that are more natural in this framework are those that are more relevant to the practice of programming.

References

[1] Ben-Amram, A. M. and Jones, N. D.: "Computational complexity via programming languages: constant factors do matter". Submitted for publication, 1998.

[2] Gregusova, L. and Korec, I.: "Small universal Minsky machines". *Mathematical foundations of computer science, Proc. 8th Symp.*, Olomouc/Czech, 1979, Lect. Notes Comput. Sci. 74, 308–316.

[3] Jones, N. D.: "Constant time factors do matter". *Proc. 25th Annual ACM Symp. on Theory of Computing*, 1993, 602–611.

[4] Korec, I.: "Small universal register machines", *Theor. Comput. Sci.*, **168**(2), 1996, 267–301.

[5] Minsky, M. L.: "Size and structure of universal Turing machines using tag systems". *Proc. Symp. Pure Math.* **5**, 1962, 229–238.

[6] Rogozhin, Y.: "Small universal Turing machines", *Theor. Comput. Sci.*, **168**(2), 1996, 215–240.

[7] Sudborough, H. and Zalcberg, Z.: "On families of languages defined by time-bounded random access machines", *SIAM Journal of Computing*, **5**, 1976, 217–230.

[8] Watanabe, S.: "5-symbol 8-state and 5-symbol 6-state universal Turing machines", *J. Assoc. Comput. Mach.*, **8**, 1961, 476–483.

[9] Žák, S.: "A turing machine time hierarchy (note)", *Theoretical Computer Science*, **26**, 1983, 327–333.

Appendix 1: **step.i**

```
;
; STEP macro
;
; with representation:
; X = (Cd . ( Stk . ( Cntr . Val ) ) )
; for atom encoding see Table 4 in paper
;
;(defun step (X)  ; include this to run under Common LISP

(if (hd X)      ;if there is Cd
(if (hd (hd (hd X)))
    (if (hd (hd (hd (hd X))))
        (if (hd (hd (hd (hd (hd X)))))
                (if (tl (hd (hd (hd (hd X)))))
                (if (tl (hd (hd (hd X))))
                    ;hd
                    (:= (cons (cons (tl (hd (hd X))) (cons (quote do_hd)
                        (tl (hd X)))) (tl X)))
                    ;tl
                    (:= (cons (cons (tl (hd (hd X))) (cons (quote do_tl)
                        (tl (hd X)))) (tl X)))
                )
                (if (tl (hd (hd (hd X))))
                    ;while
                    (:= (cons (cons (hd (tl (hd (hd X)))) (cons (quote do_while)
                        (hd X))) (tl X)))
                    ;cons
                    (:= (cons (cons (hd (tl (hd (hd X)))) (cons
                        (tl (tl (hd (hd X)))) (cons (quote do_cons) (tl (hd X))
                        ))) (tl X)))
                )
            )
            (if (tl (hd (hd (hd X))))
               ;semicolon
               (:= (cons (cons (hd (tl (hd (hd X)))) (cons (tl (tl (hd (hd X))))
                   (tl (hd X)))) (tl X)))
               ;:=
               (:= (cons (cons (tl (hd (hd X))) (cons (quote do_assign)
                   (tl (hd X)))) (cons (hd (tl X)) (cons
                   (hd (tl (tl X))) (tl (tl (tl X)))))))
            )
        )
        (if (tl (hd (hd (hd X))))
            ;quote
            (:= (cons (tl (hd X)) (cons (cons (tl (hd (hd X)))
                (hd (tl X))) (cons (tl (hd (tl (tl X))))
                (tl (tl (tl X)))))))
            ;if
            (:= (cons (cons (hd (tl (hd (hd X)))) (cons (quote do_if) (cons
                (hd (tl (tl (hd (hd X))))) (cons (tl (tl (tl (hd (hd X)))))
                (tl (hd X)))))) (tl X)))
        )
    )
;
;(hd (hd (hd X))) = nil, so (hd (hd X)) is the code of an atom
;
    (if (hd (tl (hd (hd X))))
        (if (tl (hd (tl (hd (hd X)))))
            (if (tl (tl (hd (hd X))))
                (if (hd (hd (tl (hd (hd X)))))
                    ;do_assgn
                    (:= (cons (tl (hd X)) (cons (tl (hd (tl X))) (cons
                        (tl (hd (tl (tl X)))) (hd (hd (tl X)))))))
                    ;X
                    (:= (cons (tl (hd X)) (cons (cons (tl (tl (tl X)))
                        (hd (tl X))) (cons (tl (hd (tl (tl X))))
                        (tl (tl (tl X)))))))
                )
                ;do_hd
                (:= (cons (tl (hd X)) (cons (cons (hd (hd (hd (tl X))))
                    (tl (hd (tl X)))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
            )
            (if (tl (tl (hd (hd X))))
                ;do_tl
                (:= (cons (tl (hd X)) (cons (cons (tl (hd (hd (tl X))))
                    (tl (hd (tl X)))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
                ;do_while
                (if (hd (hd (tl X))) (:= (cons (cons (tl (tl (hd (tl (hd X)))))
                    (tl (hd X))) (cons (tl (hd (tl X)))
                    (cons (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
                    (:= (cons (tl (tl (hd X)))
                    (cons (tl (hd (tl X))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
                )
            )
        )
        (if (tl (tl (hd (hd X))))
            ;do_cons
            (:= (cons (tl (hd X)) (cons (cons (cons (hd (tl (hd (tl X))))
                (hd (hd (tl X)))) (tl (tl (hd (tl X)))))
                (cons (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
            ;do_if
            (if (hd (hd (tl X))) (:= (cons (cons (hd (tl (hd X)))
                (tl (tl (tl (hd X))))) (cons (tl (hd (tl X))) (cons
                (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
                (:= (cons (cons (hd (tl (tl (hd X))))
                    (tl (tl (tl (hd X))))) (cons (tl (hd (tl X))) (cons
                    (tl (hd (tl (tl X)))) (tl (tl (tl X))))))))
        )
    )
)
;if Cd is empty
(:= (cons (quote nil) (cons (quote nil) (cons (quote nil) (tl (tl (tl X)))))))
)
;) ;defun     ;include this to run under Common LISP
```

Appendix 2: **tree-size.i**

```
;       tree-size.i
;      =============
;
; this is essentially the tree-sizing program from [1].
; the two variables of the program (X,Y) are represented here in a single
; variable as (X.Y.Z) where Z is a copy of the input, preserved for use
; by the rest of the program.
; The program starts with the input in X and the first assignment builds
; the above triple.
;

; I language definitions for running under LISP (more complex than in step
; because we need imperative features here).

(defmacro hd (expr) `(car ,expr))
(defmacro tl (expr) `(cdr ,expr))
(defmacro := (expr) `(setq X ,expr))
(defmacro $ (C1 C2) `(progn ,C1 ,C2 X))
(defmacro while (test body) `(do () ((not ,test) X) ,body))


(defun size (X)
($
  (:= (cons X (cons (quote (nil)) X)))                 ;TIME = 6
  (while (hd X)                                        ;TIME = 3
    (if (hd (hd X))
        (:= (cons (cons (hd (hd (hd X))) (cons (tl (hd (hd X))) (tl (hd X))))
          (tl X)))                                     ;TIME = 21
        (:= (cons (tl (hd X))
            (cons (cons (quote nil) (cons (quote nil) (hd (tl X))))
            (tl (tl X))                                ;TIME = 20
        )))
    )
  )
)
)
```