Constant Time Factors *Do* Matter

(Extended abstract)

Neil D. Jones*

Abstract

The *constant speedup theorem*, so well known from Turing machine based complexity theory, is shown false for a natural imperative programming language `I` that manipulates tree-structured data. This relieves a tension between general programming practice, where linear factors are essential, and complexity theory, where linear time changes are traditionally regarded as trivial.

Specifically, there is a constant b such that for any $a > 0$ there is a set X recognizable in time[1] $a \cdot b \cdot n$ but not in time $a \cdot n$. Thus **LIN**, the collection of all sets recognizable in linear time by deterministic `I`-programs, contains an infinite hierarchy ordered by constant coefficients. Constant hierarchies also exist for larger time bounds $T(n)$, provided they are time-constructable.

Second, a problem is exhibited which is complete for the nondeterministic linear time sets **NLIN** with respect to a natural notion of deterministic linear-time reduction.

Third, Kleene's Second Recursion Theorem in essence shows that for any program `p` defined with self-reference, there is an equivalent nonreflexive program `q`. This is proven for an extension $\mathtt{I}^\uparrow$ of `I`. Further, `q` can be simulated by an `I` program at most constantly slower than `p`. Language $\mathtt{I}^\uparrow$ allows calls to the language's own interpretation function, and even to its running time function (without the usual high costs for nested levels of interpretation).

The results all hold as well for a stronger language $\mathtt{I}^{su}$ allowing selective updating of tree-structured data. The results are robust in that classes $\mathbf{LIN}^{su}$ and $\mathbf{NLIN}^{su}$ are identical for $\mathtt{I}^{su}$, $\mathtt{I}^{su\uparrow}$, Schönhage's Storage Modification Machines, Knuth/Tarjan's pointer machines, and successor RAMs[2] [13,14,11].

[1]where n is the size of the input.

[1]DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, E-mail: neil@diku.dk.

[2]If the "more realistic and precise measure" of SMM computation time is used [13], and similarly for the other models.

1 Introduction

Our thesis is that the constant speedup theorem for Turing machines is an artifice of the model, due to the assumption that a Turing machine's tape alphabet can be arbitrarily large. The thesis will be supported by introducing an alternative computation model, an imperative programming language **I** which has a "fair accounting" of computation time.

Gurevich and Shelah have pointed out several problems in defining *linear time* computations using Turing machines, and resolved them by showing that the class NLT of "nearly linear time" functions is quite robust with respect to the choice of machine model [5].

Our new model allows a natural definition of linear time computation, and it leads to a proper hierarchy based on constant multiplicative factors. The class is seen to be robust in the sense that imperative and functional versions of the language define the same class of linear time solvable problems. Further, an extension $\mathbf{I}^{su}$ of the language with "selective updating" is seen equivalent to several machine models well-known from the literature, including most of those mentioned in [5].

The key to most of our results is the existence of an "efficient" universal program, capable of simulating any program `p` at most a times slower than `p` itself runs, where the slowdown factor a is independent of `p`.

2 Syntax, semantics, and running time for `I`

2.1 Data

Input and output data values are Lisp list structures formed from pairing and a single atom `nil`: the least set satisfying $D = \{\mathtt{nil}\} \cup D \times D$. The list construction ("cons") operator is "`.`". There are corresponding first and second component projections, e.g. *head*(`(a.(b.c))`) = `a` and *tail*(`(a.(b.c))`) = `(b.c)`).

Pairing is right associated, so `(a.b.c)`=`(a.(b.c))` (written using several atoms for easier reading by humans). A Lisp-like notation with implicit constructors is used for compactness, for example `(b c)` stands for `(b.(c.nil))`, and `(a (b c) d)` stands for `(a.((b.(c.nil)).(d.nil)))`.

The *size* of $\mathtt{d} \in D$ is defined inductively: $|\mathtt{d}| = 1$ if `d` is an atom, and $|\mathtt{d1}\,.\,\mathtt{d2}| = |\mathtt{d1}| + |\mathtt{d2}|$.

Syntactic category:			*Informal syntax:*	*Concrete syntax:*
`P` : Program	`::=`		`read X; C; write X`	`C`
`C` : Command	`::=`		`X := E`	`(:= E)`
		$\mid$	`C;C`	`(; C C)`
		$\mid$	`if E then C else C`	`(if E C C)`
		$\mid$	`while E do C`	`(while E C)`
`E` : Expression	`::=`		`X`	`X`
		$\mid$	$'\texttt{D}$	`(quote D)`
		$\mid$	`hd E`	`(hd E)`
		$\mid$	`tl E`	`(tl E)`
		$\mid$	$\texttt{E . E}'$	$\texttt{(cons E E}'\texttt{)}$
`X` : Variable	`::=`		`x`	
`D` : Data-value	`::=`		$\texttt{A} \mid \texttt{D.D}$	
`A` : Atom	`::=`		`nil`	

Figure 1: Program syntax: informal and concrete

$[\![\texttt{S}]\!] : D \to D \cup \{\bot\}$	$t_{\texttt{S}} : D \to \mathcal{N} \cup \{\infty\}$
$[\![\texttt{x := E}]\!]\,\texttt{d} = [\![\texttt{E}]\!]\,\texttt{d}$	$t_{\texttt{x:=E}}(\texttt{d}) = 1 + t_{\texttt{E}}(\texttt{d})$
$[\![\texttt{C1;C2}]\!]\,\texttt{d} = [\![\texttt{C2}]\!]([\![\texttt{C1}]\!]\,\texttt{d})$	$t_{\texttt{C1;C2}}(\texttt{d}) = t_{\texttt{C1}}(\texttt{d}) + t_{\texttt{C2}}([\![\texttt{C1}]\!]\,\texttt{d})$
$[\![\texttt{if E then C1 else C2}]\!]\,\texttt{d} =$	$t_{\texttt{ifEthenC1elseC2}}(\texttt{d}) = 1 + t_{\texttt{E}}(\texttt{d}) +$
$\quad \textit{if } [\![\texttt{E}]\!]\,\texttt{d} \neq \texttt{nil} \textit{ then } [\![\texttt{C1}]\!]\,\texttt{d} \textit{ else } [\![\texttt{C2}]\!]\,\texttt{d}$	$\quad \textit{if } [\![\texttt{E}]\!]\,\texttt{d} \neq \texttt{nil} \textit{ then } t_{\texttt{C1}}(\texttt{d}) \textit{ else } t_{\texttt{C2}}(\texttt{d})$
$[\![\texttt{while E do C}]\!]\,\texttt{d} = (\textit{if } [\![\texttt{E}]\!]\,\texttt{d} = \texttt{nil} \textit{ then } \texttt{d}$	$t_{\texttt{whileEdoC}}(\texttt{d}) = 1 + t_{\texttt{E}}(\texttt{d}) +$
$\quad \textit{else } [\![\texttt{while E do C}]\!]([\![\texttt{C}]\!]\texttt{d}))$	$\quad \textit{if } [\![\texttt{E}]\!]\,\texttt{d} = \texttt{nil} \textit{ then } 0 \textit{ else } t_{\texttt{whileEdoC}}([\![\texttt{C}]\!]\,\texttt{d})$
$[\![\texttt{x}]\!]\,\texttt{d} = \texttt{d}$	$t_{\texttt{x}}(\texttt{d}) = 1$
$[\![\texttt{'d1}]\!]\,\texttt{d} = \texttt{d1}$	$t_{\texttt{'nil}}(\texttt{d}) = 1,$ and $t_{\texttt{'(d1.d2)}}(\texttt{d}) = t_{\texttt{d1}}(\texttt{d}) + t_{\texttt{d2}}(\texttt{d})$
$[\![\texttt{hd E}]\!]\,\texttt{d} = (\textit{if } \texttt{(t.u)} = [\![\texttt{E}]\!]\,\texttt{d} \textit{ then } \texttt{t} \textit{ else } \texttt{nil})$	$t_{\texttt{hdE}}(\texttt{d}) = 1 + t_{\texttt{E}}(\texttt{d})$
$[\![\texttt{tl E}]\!]\,\texttt{d} = (\textit{if } \texttt{(t.u)} = [\![\texttt{E}]\!]\,\texttt{d} \textit{ then } \texttt{u} \textit{ else } \texttt{nil})$	$t_{\texttt{tlE}}(\texttt{d}) = 1 + t_{\texttt{E}}(\texttt{d})$
$[\![\texttt{E1 . E2}]\!]\,\texttt{d} = ([\![\texttt{E1}]\!]\,\texttt{d}\ .\ [\![\texttt{E2}]\!]\,\texttt{d})$	$t_{\texttt{E1.E2}}(\texttt{d}) = 1 + t_{\texttt{E1}}(\texttt{d}) + t_{\texttt{E2}}(\texttt{d})$

Figure 2: Program meanings and running times in imperative language `I`

2.2 Syntax of language `I`

Figure 1 defines both an informal syntax used to describe programs for reading, and a concrete syntax, used to present programs as data objects for processing by other programs.

Programs have only one variable `x`, and manipulate tree structures built from the one atom `nil`. Generalization to many variables and atoms will be discussed later. In `if` and `while` tests, value `nil` is considered *false* and any other value is considered *true*.

2.3 Semantics and time usage

Definition 2.1 *The semantic function* $[\![\texttt{p}]\!] : D \to D \cup \{\bot\}$ *and running time* $t_{\texttt{p}}(\texttt{d}) : D \to \mathcal{N} \cup \{\infty\}$ *of program* `p` *are the least functions that satisfy Figure 2. Variable* `S` *ranges over commands and expressions.*

Here $D \cup \{\bot\}$ is partially ordered by $\bot \leq \texttt{d}$ for all `d`, and $\mathcal{N} \cup \{\infty\}$ is ordered by $n \leq \infty$ for all n (with no other relations). Functions are ordered pointwise.

Informally $[\![\texttt{p}]\!]\,\texttt{d} = \bot$ indicates nontermination, as does $t_{\texttt{p}}(\texttt{d}) = \infty$. The time measure of Figure 2 accords with standard Lisp implementation on a "unit-cost RAM", and so is faithful to current programming practice. Time measures will be discussed in more detail in Section 8.

2.4 Variations on the language

We first show how to extend this tiny language `I` without essentially affecting its run time. Some (but not all) extended programs may be translated into `I` programs that run slower, but at most by a constant factor *independent* of the program being translated.

2.4.1 More but uniformly bounded numbers of atoms and variables

First, we show that `I`-programs can simulate programs with many variables or atoms, and with at worst a constant slowdown if there is a uniform bound on the numbers of variables and atoms.

More atoms. Assume for example that a given program uses atoms `0` and `1` in addition to `nil`, so D is extended to $D' = \{\texttt{nil}, \texttt{0}, \texttt{1}\} \cup D' \times D'$. The extended syntax and semantics are the same except that the atom set is larger,

$$\begin{array}{lll}
Acc(\texttt{p}) & = & \{\texttt{d} \in D \quad \mid \bot \neq [\![\texttt{p}]\!]\,\texttt{d} \neq \texttt{nil}\} \\
\mathbf{LIN}(a) & = & \{Acc(\texttt{p}) \quad \mid \forall \texttt{d} \in D\,.\; t_{\texttt{p}}(\texttt{d}) \leq a \cdot |\texttt{d}|\} \\
\mathbf{LIN} & = & \cup_{a \geq 0}\mathbf{LIN}(a) \\
 & & \\
Acc_{\exists}(\texttt{p}) & = & \{\texttt{d} \in D \quad \mid \exists \texttt{d}^{or} \in D\,.\; (\texttt{d.d}^{or}) \in Acc(\texttt{p})\} \\
\mathbf{NLIN}(a) & = & \{Acc_{\exists}(\texttt{p}) \quad \mid \forall\ \texttt{d} \in D\,.\,\texttt{d} \in Acc_{\exists}(\texttt{p}) \text{ implies} \\
 & & \qquad \exists\ \texttt{d}^{or}\,.\,(\texttt{d.d}^{or}) \in Acc(\texttt{p}) \text{ and } t_{\texttt{p}}((\texttt{d.d}^{or})) \leq a \cdot |\texttt{d}|\} \\
\mathbf{NLIN} & = & \cup_{a \geq 0}\mathbf{NLIN}(a)
\end{array}$$

Figure 3: Complexity class definitions

and that tests of form `x = 'd` are allowed in `if` and `while` commands, with the natural meanings.

A list structure `d` in D' may be encoded as an element $\overline{\texttt{d}}$ of the original D by letting $\overline{\texttt{d1.d2}} = \overline{\texttt{d1}}\ .\ \overline{\texttt{d2}}$ and

$$\overline{\texttt{nil}} = \texttt{nil},\ \overline{0} = \texttt{nil.nil},\ \overline{1} = \texttt{nil.nil.nil}$$

Given this representation, $\left|\overline{\texttt{d}}\right| \leq 3 + 2 \cdot |\texttt{d}|$, and the operations `hd`, `tl` and `cons` can all be simulated in constant time. For any $\texttt{d} \in D'$ it is clear that the test "`x = 'd`?" can be simulated in time $O(|\texttt{d}|)$ by nested `if`'s.

More variables. Consider a program with three variables `Cd`, `Stk`, `Val` (as we will use later). One can convert it to an equivalent 1-variable program by "packing" the three into one: `x = Cd.Stk.Val = (Cd.(Stk.Val))`. By unpacking and packing together again, command `Stk:=Val.Stk` (for instance) can be realized with code expansion and slowdown determined only by the number of variables to be represented as follows:

```
x := (hd x).((tl tl x).(hd tl x)).(tl tl x)
```

2.4.2 Extensions with program dependent slowdown

Suppose we break uniform boundedness and allow programs with unboundedly many atomic symbols (as in Turing machines or in Lisp) and numbers of variables (allowed by Lisp). Any such program can be translated into one with only one variable and one atom by the technique just given — but the slowdown factor will depend on the program being translated. (We will in fact use this extension later.)

2.5 Turing machine simulation in real time

Theorem 2.2 *For any k-tape Turing machine* `Z` *there is an* `I`*-program* `p` *and constant a such that for any bit list* `d` *(with $\overline{\texttt{d}}$ as above)*

$$[\![\texttt{p}]\!]\,\overline{\texttt{d}} = \overline{[\![\texttt{Z}]\!]_{TM}\,\texttt{d}} \quad \text{and} \quad t_{\texttt{p}}(\overline{\texttt{d}}) \leq a \cdot t^{TM}_{\texttt{Z}}(\texttt{d})$$

Proof is by a very straightforward construction. Note that constant a will depend on the Turing machine `Z`.

2.6 Definition of complexity classes

Figure 3 defines the deterministic and nondeterministic classes **LIN** and **NLIN**. Nondeterminism is expressed by giving the program an extra input $\texttt{d}^{or}$ as an "oracle", to guide program execution by angelic choice. Note that the time bound must be linear in the size of `d`, regardless of the size of $\texttt{d}^{or}$.

We will prove that the deterministic hierarchy is proper, and exhibit a complete problem for **NLIN**. The results can be extended to general time classes **DTIME**$(T(n))$ and **NTIME**$(T(n))$, omitted here for brevity's sake.

3 Efficient universal programs

The Turing machine's unbounded tape alphabet is the source of the "constant speedup theorem" [6], which we will prove false for our name- and atom-bounded computing model. The key is the existence of an efficient universal program `u` (self-interpreter) for `I` programs. Although `u` runs slower than the program being interpreted, it is slower only by a multiplicative factor *independent of the program being simulated.*

Definition 3.1 `u` *is a* universal program *if for every* `d` $\in D$ *and program* `p`

$$[\![\texttt{u}]\!]\ (\texttt{p . d}) = [\![\texttt{p}]\!]\ \texttt{d}$$

Definition 3.2 *Universal program* `u` *is* efficient *if there is a constant a such that for all* $\texttt{d} \in D$ *and program* `p`

$$t_{\texttt{u}}((\texttt{p . d})) \leq a \cdot t_{\texttt{p}}(\texttt{d})$$

Note that constant a is quantified *before* `p`, so the slowdown caused by an "efficient" interpreter is independent of `p`.

3.1 A universal program

For readability's sake we use some extra atoms (`if`, `:=`, ...). As above, they are abbreviations for structures built from `nil` and "`.`".

The universal program `u` is the one-atom, one-variable translation of the following four variable program. To aid compactness and readability, we describe the `STEP` part of the algorithm by a set of transitions, i.e. rewrite rules. Blank entries in Figure 4 correspond to values that are neither referenced nor changed in a rule.

```
read x;
Cd := (hd x) . 'nil;   (* Code to be executed *)
Val := tl x;           (* Initial value of simulated x *)
Stk := 'nil;           (* Computation stack *)
while Cd do STEP;
x := Val;              (* Val is final value of x *)
write x
```

Code	Stack	Value	$\Rightarrow$	Code	Stack	Value
`nil`			$\Rightarrow$	`nil`		
`(';  C1 C2).Cd`			$\Rightarrow$	`C1.C2.Cd`		
`(':= E).Cd`			$\Rightarrow$	`E.'assign.Cd`		
`'assign.Cd`	`w.s`	`v`	$\Rightarrow$	`Cd`	`s`	`w`
`('if E C1 C2).Cd`			$\Rightarrow$	`E.'do_if.C1.C2.Cd`		
`'do_if.C1.C2.Cd`	`(t.u).s`		$\Rightarrow$	`C1.Cd`	`s`	
`'do_if.C1.C2.Cd`	`nil.s`		$\Rightarrow$	`C2.Cd`	`s`	
`('while E C).Cd`			$\Rightarrow$	`E.'do_while.` `('while E C).Cd`		
`'do_while.` `('while E C).Cd`	`(t.u).s`		$\Rightarrow$	`C.('while E C).Cd`	`s`	
`'do_while.C1.Cd`	`nil.s`		$\Rightarrow$	`Cd`	`s`	
`'x.Cd`	`s`	`v`	$\Rightarrow$	`Cd`	`v.s`	`v`
`'nil.Cd`	`s`		$\Rightarrow$	`Cd`	`nil.s`	
`('hd E).Cd`			$\Rightarrow$	`E.'do_hd.Cd`		
`'do_hd.Cd`	`(t.u).s`		$\Rightarrow$	`Cd`	`t.s`	
`'do_hd.Cd`	`nil.s`		$\Rightarrow$	`Cd`	`nil.s`	
`('tl E).Cd`			$\Rightarrow$	`E.'do_tl.Cd`		
`'do_tl.Cd`	`(t.u).s`		$\Rightarrow$	`Cd`	`u.s`	
`'do_tl.Cd`	`nil.s`		$\Rightarrow$	`Cd`	`nil.s`	
`('cons E1 E2).Cd`			$\Rightarrow$	`E1.E2.'do_cons.Cd`		
`'do_cons.Cd`	`u.t.s`		$\Rightarrow$	`Cd`	`(t.u).s`	

Figure 4: The `STEP` *macro, expressed by rewriting rules*

For example the three `while` transitions could be programmed as:

```
if hd hd Cd = 'while then (* Set up iteration *)
    Cd := (hd tl hd Cd) . 'do_while . Cd
else
if hd Cd = 'do_while then
    if hd Stk then (* Do body if test is true *)
    { Cd := (hd tl tl tl Cd) . (tl Cd);
      Stk := tl Stk } else (* Else exit while *)
    { Stk := tl Stk; Cd := tl tl Cd }
else ...
```

Theorem 3.3 `u` *is an efficient universal program.*

PROOF: The interpreter uses standard techniques, so correctness is easily verified. As to time, first note that the entire `STEP` macro is a fixed piece of noniterative code, so the commands to find one matching transition in the table and realize its effect take constant time independent of interpreted program `p`.

Further, any single step of the interpreted program is realized in a bounded amount of time. For example, the decision of whether to iterate a `while` loop takes one step in `p`. It is realized interpretively by one transition to set up the code stack before evaluating the expression to be tested, a test done afterwards to see whether to iterate or not, and then the chosen transition is applied. In this way a uniform bound on the interpretation/execution time ratio exists for *all* computations. □

3.2 A timed universal program

Definition 3.4 *Let* $\texttt{nil}^n$ = `(nil nil ...nil)`*, i.e. the length n list of* `nil`*'s. Then* `tu` *is an* efficient timed universal program *if there is a constant k such that for all* `p`, `d` $\in D, n \geq 1$*:*

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

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

$t_{\texttt{tu}}((\texttt{p . d . nil}^n)) \leq k \cdot min(n, t_{\texttt{p}}(\texttt{d}))$

To construct such a program using the STEP macro we only change the main program to:

```
read x;
Cd := (hd x) .  'nil;     (* Code to be executed *)
Val := hd tl x;           (* Init. value, simulated x *)
Cntr := tl tl x;          (* Time bound *)
Stk := 'nil;              (* Computation stack *)
while Cd do
  if Cntr then {STEP; Cntr := tl Cntr}
  else {Cd := 'nil; Stk := '(nil.nil)};
x := Val;                 (* Final x value *)
write x
```

Theorem 3.5 tu *is an efficient timed universal program.*

3.3 A nondeterministic universal program

This can again be obtained by a small modification to the universal program u. By definition input d is in $Acc_\exists$(p) iff (d.d^{or}) is in Acc(p) for some d^{or}. Consider the following modification "nu" of u:

```
read x;
Cd := (hd hd x) .  'nil; (* Input ((p.d).d^or) *)
Val:=(tl hd x).(tl x);   (* Simulated x = (d.d^or) *)
Stk := 'nil;             (* Computation stack *)
while Cd do STEP;
x := Val;                (* Final x value *)
write x
```

It is easy to see that nu is efficient as we have defined the term, and that (p.d)$\in Acc_\exists$(nu) iff d $\in Acc_\exists$(p).

4 Constant time factors give a proper hierarchy

Theorem 4.1 *There is a b such that for all $a > 0$,* **LIN**$(a \cdot b) \setminus$ **LIN**(a) *is non-empty.*

PROOF: First define program diag informally:

```
read x;
Timebound := nil^(a·|x|);
x := tu(x.x.Timebound);
if x then x := 'nil else x := '(nil.nil);
write x
```

Since a is fixed, nil$^{a\cdot|\mathtt{x}|}$ can be computed in time $c \cdot |\mathtt{x}|$ for some c. From Definition 3.4, there exists k such that the timed universal program tu of Theorem 3.5 runs in time $t_{\mathtt{tu}}$((p . d. nil^n)) $\leq k \cdot min(n, t_{\mathtt{p}}(\mathtt{d}))$. To define diag formally, replace x := tu(...) by

```
x := x.x.Timebound; Body;
```

(Body is the part of tu between read x and write x).

On input p, program diag runs in time at most $e+(c+k)\cdot a\cdot|\mathtt{p}|$ where c, k are constants as above, and e accounts for the time beyond computing Timebound and running tu. Defining $b = c + k + e$, we have shown Acc(diag) $\in$ **LIN**$(a \cdot b)$.

Now suppose for the sake of contradiction that Acc(diag) $\in$ **LIN**(a). Then there exists a program p with Acc(p) $= Acc$(diag), and $t_{\mathtt{p}}(\mathtt{d}) \leq a \cdot |\mathtt{d}|$ for all d $\in D$. Now consider cases of $[\![\mathtt{p}]\!]\,\mathtt{p}$ (the usual diagonal argument). The time bound on p implies that $[\![\mathtt{tu}]\!]$(p.p.nil$^{a\cdot|\mathtt{p}|}$) $= [\![\mathtt{p}]\!]\,\mathtt{p}$. If this is nil, then $[\![\mathtt{diag}]\!]\,\mathtt{p}$ = (nil.nil). If it is not nil, then $[\![\mathtt{diag}]\!]\,\mathtt{p}$ = nil. Both cases contradict Acc(p) $= Acc$(diag), so the assumption that Acc(diag) $\in$ **LIN**(a) must be false. □

Further, for any non-zero "time constructable" $T(n)$ (using the natural definition), there is a b such that $DTIME(b \cdot T(n)) \setminus DTIME(T(n))$ is non-empty.

5 A complete problem for NLIN

We now show that **NLIN** has a "hardest" problem, in analogy with the much-studied class **NP** [6].

Definition 5.1 *$f : D \to D$ is* linear time computable *if there are $a, b,$ p such that $f = [\![\mathtt{p}]\!]$, and $t_{\mathtt{p}}(\mathtt{d}) \leq a \cdot |\mathtt{d}|$, and $|f(\mathtt{d})| \leq b \cdot |\mathtt{d}|$ for all d $\in D$.*

Definition 5.2 *Let $L, M \subseteq D$. Then L is* reducible *to M (written $L \leq M$) iff there is a linear time computable function f such that* d $\in L$ iff $f(\mathtt{d}) \in M$ *for all* d *in D.*

Further, $P \subseteq D$ is complete *for* **NLIN** *iff $P \in$* **NLIN** *and $L \leq P$ for all $L \in$* **NLIN**.

Lemma 5.3 $\leq$ *is a reflexive and transitive relation.*

Lemma 5.4 $L \leq M$ *and* $M \in$ **LIN** *implies* $L \in$ **LIN**.

Corollary 5.5 *If P is complete for* **NLIN**, *then $P \in$* **LIN** *if and only if* **NLIN** = **LIN**.

Theorem 5.6 NU_0 *is complete for* **NLIN**, *where*

$$NU_0 = \{(\mathtt{p.d}) \mid \mathtt{p} \textit{ is } \mathtt{while}\textit{-free and } \mathtt{d} \in Acc_\exists(\mathtt{p})\}$$

PROOF: To show $NU_0 \in$ **NLIN**, modify the nondeterministic universal program nu as follows. First, check that input p is an encoded while-free program, and then run nu. The checking can be done in time linear in |p|, and while-freeness implies that $t_{\mathtt{p}}(\mathtt{d}) \leq |\mathtt{p}|$ regardless of d. Thus recognition of NU_0 takes time at most linear in |(p . d)| for all p, d in D.

Now suppose $L = Acc_\exists(\mathtt{p}) \in$ **NLIN**(a). Given d, define $f(\mathtt{d})$ = (q . d) where q is the following program, and STEP$^{a\cdot|\mathtt{d}|}$ stands for $a \cdot |\mathtt{d}|$ copies of the code for the STEP macro:

```
read x;
Cd := 'p .  'nil;        (* Code to be executed *)
Val := x;                (* Initial input (d.d^or) *)
Stk := 'nil;             (* Computation stack *)
STEP^(a·|d|);            (* a · |d| = time bound *)
x := Val;
write x
```

Clearly f is linear time computable. Program q is while-free, and it works by simulating p on d for $a \cdot |\mathtt{d}|$ steps. This is long enough to produce p's output, so d $\in L$ iff d $\in Acc_\exists$(p) iff d $\in Acc_\exists$(q) iff $f(\mathtt{d}) \in NU_0$.
□

6 Efficient programs from Kleene's recursion theorem

Kleene's *second recursion theorem* [10], valid for any "acceptable enumeration" [12] of the partial recursive functions, guarantees the computability of functions defined by self-referential algorithms. It has many applications in recursive function theory, machine-independent computational complexity, and learning theory [3,4]. We show how to obtain much more efficient programs than those given by traditional constructions [10,12].

Our technique is similar to that of [1], but the use of an "efficient" universal program strengthens the results shown there by giving faster programs and explicit complexity bounds. First some motivating discussion (readers less interested in recursive functions may skip to the Section on robustness, as the following results are not used later).

6.1 The theorem and applications

Our formalism provides an acceptable enumeration, and Kleene's theorem may be stated as:

Theorem 6.1 *For any program* p*, there is a program* q *such that for all inputs* d*,*

$$[\![\mathtt{q}]\!]\,\mathtt{d} = [\![\mathtt{p}]\!]\;\mathtt{(q.d)}$$

Typically p's first input is a program, which p may apply to various arguments, transform, time, or process as it sees fit. The theorem in effect says that p may regard q is *its own text*, thus allowing self-referential programs. Before proving an efficient version, we give two applications using an extended language $\mathtt{I}^{\uparrow}$:

Example 1. Elimination of recursion is illustrated by letting p be the following, where for readability names r, y have been introduced for resp. hd x, tl x. Construction "run E1 on E2" evaluates E1 and E2, and treats the first value as a program which is applied to the second value. Thus the program transforms $\mathtt{y} = \mathtt{d}_1\mathtt{.}\ldots\mathtt{.d}_n\mathtt{.nil}$ into $\mathtt{nil.d}_n\mathtt{.}\ldots\mathtt{.d}_1$.

```
read x; (* x ⇒ (r.y) *)
if y
then x := (run r on r.(tl y)) . (hd y)
else x := 'nil;
write x
```

By Kleene's theorem there is a *nonrecursive* program q such that

$$[\![\mathtt{q}]\!](\mathtt{d}_1\mathtt{.}\ldots\mathtt{.d}_n\mathtt{.nil}) = \mathtt{nil.d}_n\mathtt{.}\ldots\mathtt{.d}_1$$

A conclusion is that language $\mathtt{I}^{\uparrow}$ is "closed under recursion". Another easy example extends a simple theorem from [3], where we represent integer n by $\mathtt{nil}^n$.

Theorem 6.2 *For any two total recursive functions* f, h *there is a program* q *such that* $[\![\mathtt{q}]\!] = f$ *and* $t_{\mathtt{q}}(\mathtt{y}) > h(\mathtt{q.y})$ *for all* $\mathtt{y} \in D$*.*

Proof: Recursivity of functions f, h implies they are computable by I programs. Consider the following $\mathtt{I}^{\uparrow}$ program p:

```
read x; (* x ⇒ (r.y) *)
bound := h(x);      (* Assumed computable *)
test := time(r,y)≤bound;
if test then x := 'nil.(run r on y)
        else x := f(y); (* Assumed computable *)
write x
```

The construction "time(r,y)≤bound" compares $t_{\mathtt{r}}(\mathtt{y})$ with bound (it will be shown efficiently computable in the next subsection). Function $[\![\mathtt{p}]\!]$ is total since $t_{\mathtt{r}}(\mathtt{y}) \leq$ bound implies $[\![\mathtt{r}]\!]\,\mathtt{y}$ terminates. By Kleene's theorem there is a program fix such that for all y, $[\![\mathtt{fix}]\!]\,\mathtt{y} = [\![\mathtt{p}]\!](\mathtt{fix.y})$. Now suppose $t_{\mathtt{fix}}(\mathtt{y}) \leq h(\mathtt{fix.y})$ for some y. Then

$$[\![\mathtt{fix}]\!]\,\mathtt{y} = [\![\mathtt{p}]\!](\mathtt{fix.y}) = \mathtt{nil.}([\![\mathtt{fix}]\!]\,\mathtt{y})$$

which is a contradiction. Thus $t_{\mathtt{fix}}(\mathtt{y}) > h(\mathtt{fix.y})$ for all y, as required. □

6.2 A reflexive language

Let $\mathtt{I}^{\uparrow}$ be the language I already seen, extended by the constructions seen above. The semantic and running time functions $[\![\_]\!]^{\uparrow}$, $t^{\uparrow}_{\mathtt{p}}$ for the new constructions are defined in Figure 5 where p is the text of the program being executed, and all other meanings and timings are as in Figure 2. The times $t^{\uparrow}_{\mathtt{p}}$ are as small as one could reasonably hope for, and we will see that they are computationally realistic.

The value of "time(E1,E2) ≤ E3" is *true* (nil.nil) if t's computation terminates within |x| steps, else *false* (nil). The value of constant * is the program currently being executed.

Lemma 6.3 *There exists an efficient interpreter* u↑ *for* $\mathtt{I}^{\uparrow}$ *programs, written in* I*.*

Proof: Efficiency in this case means

$$\exists a \forall \mathtt{p} \forall \mathtt{d}.\;\; t_{\mathtt{u}\uparrow}(\mathtt{d}) \leq a \cdot t^{\uparrow}_{\mathtt{p}}\,\mathtt{d}$$

Interpreter u↑ is built by a simple extension of u and tu to deal with the new forms. The value of * is the program currently being interpreted — available when interpretation begins, so a copy can be maintained in an auxiliary variable.

Construction "run E1 on E2" is handled by evaluating E1 and E2, and then calling u on their values.

The value of expression "time(E1,E2) ≤ E3" is obtained by evaluating E1, E2, E3 to values v, w, x, and then applying tu′ — a minor variant of the timed self-interpreter — to $(\mathtt{v.w.nil}^{|\mathtt{x}|})$. The effect of tu′ is to return *true* if program v terminates on input w within at most |x| steps, and *false* otherwise. □

Theorem 6.4 *Efficient Kleene's theorem: for any* p *there is an* $\mathtt{I}^{\uparrow}$*-program* q *and constant* a *such that for all* $\mathtt{d} \in D$, $[\![\mathtt{q}]\!]^{\uparrow}\,\mathtt{d} = [\![\mathtt{p}]\!]^{\uparrow}(\mathtt{q.d})$*, and* $t^{\uparrow}_{\mathtt{q}}(\mathtt{d}) \leq a \cdot t^{\uparrow}_{\mathtt{p}}((\mathtt{q.d}))$*.*

Proof: If q is as follows, it is immediate that $[\![\mathtt{q}]\!]^{\uparrow}\,\mathtt{d} = [\![\mathtt{p}]\!]^{\uparrow}\,(\mathtt{q.d})$. The time result follows from the definitions in Figures 5 and 2, and the "efficiency" of u and tu′.

```
  read x;
  x := (run 'p on (*.x));
  write x
```

$[\![\mathtt{S}]\!]^{\uparrow} : D \to D \cup \{\bot\}$	$t^{\uparrow}_{\mathtt{S}} : D \to \mathcal{N} \cup \{\infty\}$
$[\![\mathtt{run\ E1\ on\ E2}]\!]^{\uparrow}\,\mathtt{d} = [\![\mathrm{v}]\!]^{\uparrow}\ \mathtt{w}$ *where* $\mathrm{v} = [\![\mathrm{E1}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{w} = [\![\mathrm{E2}]\!]^{\uparrow}\,\mathtt{d}$	$t^{\uparrow}_{\mathtt{runE1onE2}}(\mathtt{d}) = 1 + t^{\uparrow}_{\mathtt{E1}}(\mathtt{d}) + t^{\uparrow}_{\mathtt{E2}}(\mathtt{d}) + t^{\uparrow}_{\mathtt{v}}(\mathtt{w})$ *where* $\mathrm{v} = [\![\mathrm{E1}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{w} = [\![\mathrm{E2}]\!]^{\uparrow}\,\mathtt{d}$
$[\![\mathtt{time(E1,E2)}\leq\mathtt{E3}]\!]^{\uparrow}\,\mathtt{d} =$ *if* $t_{\mathtt{v}}(\mathtt{w}) \leq \mathtt{x}$ *then* $\mathtt{nil.nil}$ *else* $\mathtt{nil}$, *where* $\mathrm{v} = [\![\mathrm{E1}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{w} = [\![\mathrm{E2}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{x} = [\![\mathrm{E3}]\!]^{\uparrow}\,\mathtt{d}$	$t^{\uparrow}_{\mathtt{time(E1,E2)}\leq\mathtt{E3}}(\mathtt{d}) = 1 + t^{\uparrow}_{\mathtt{E1}}(\mathtt{d}) + t^{\uparrow}_{\mathtt{E2}}(\mathtt{d}) + t^{\uparrow}_{\mathtt{E3}}(\mathtt{d}) + min(\mathtt{x}, t^{\uparrow}_{\mathtt{v}}(\mathtt{w}))$, *where* $\mathrm{v} = [\![\mathrm{E1}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{w} = [\![\mathrm{E2}]\!]^{\uparrow}\,\mathtt{d}$, $\mathrm{x} = [\![\mathrm{E3}]\!]^{\uparrow}\,\mathtt{d}$
$[\![\mathtt{*}]\!]^{\uparrow}\,\mathtt{d} = \mathtt{p}$ *The text of the enclosing program.*	$t^{\uparrow}_{*}(\mathtt{d}) = 1$

Figure 5: Extended meanings and running times

□

By Lemma 6.3, this "fixpoint program" may be efficiently simulated in I. An example application to the list reversal program above gives an I program that runs in time linear in the length of its input.

7 Robustness of the results

Let $[\![\mathtt{p}]\!]_{\mathtt{X}}\,\mathtt{d}$ and $t^{\mathtt{X}}_{\mathtt{p}}(\mathtt{d})$ denote the semantic and running time functions of programming language X, where all languages are assumed to have the same input-output set D.

Definition 7.1 *Programming language* M *can* interpret *language* L*, written* M $\succeq$ L*, if there is an* M*-program* m *such that for all* L*-programs* p *and* $\mathtt{d} \in D$

$$[\![\mathtt{m}]\!]_{\mathtt{M}}\,(\mathtt{p.d}) = [\![\mathtt{p}]\!]_{\mathtt{L}}\,\mathtt{d}$$

The interpretation is efficient *if there is an a independent of* p *and* d *such that*

$$t^{M}_{\mathtt{m}}\,(\mathtt{p.d}) \leq a \cdot t^{L}_{\mathtt{p}}(\mathtt{d})$$

This definition is stronger than the "real-time simulations" used by Schönhage and others [13] in two respects. First, we require the simulation constant to be independent of the program being simulated, which is not true of most simulations seen in the literature. Second, we require interpretability, and not just translatability from one computing formalism to another. To illustrate we consider a functional version of our language.

7.1 A functional language F

F is a simple first order Lisp-like language whose programs have one recursively defined function of one variable. We will show that F can both efficiently interpret and be interpreted by I. A consequence is

Corollary 7.2 F *has an efficient universal program. Further, the constant hierarchy theorem, and the efficient version of Kleene's Recursion Theorem both hold for* F*.*

Figures 6 and 7 define the syntax, semantics, and running times of F-programs. Auxiliary function $\mathcal{E}[\![\mathtt{E}]\!]\,\mathtt{d}\,\mathtt{B}$ gives the result of evaluating expression E with argument x = d, within the recursive function f(x) = B.

Interpretation of I by F. Multiple function arguments can be reduced to one by essentially the same packing technique as for the imperative language, and it is also easy to reduce several functions defined by mutual recursion to one function, using a "case tag" to identify the function currently being evaluated.

The following F-language interpreter for I-programs accepts an input of form (p.d) where p follows the syntax of Figure 1. Function Step realizes Figure 4 in the obvious way, so most of its details are omitted. Efficiency of the interpretation is immediate.

```
f((hd x).'nil, tl x, 'nil) whererec
  f(Cd, Val, Stk) =
    if Cd then f(Step(Cd, Val, Stk))
          else Val
  Step(Cd, Val, Stk) =
    if hd Cd = 'assign
    then Step(tl Cd, tl Stk, hd Stk)
    else if ...  (* Other transitions *)
```

Interpretation of F by I. This is done by extending the expression evaluation part of I's universal program, partly by adding new variable B that is used as in the semantic equations defining F. The interpreter has the following form, where the STEP rewrite rules are as in Figure 4 with two exceptions: the rules pertaining to commands have been removed, and three new transitions have been added to deal with function calls, detailed in Figure 8.

How they work: the call assigns to x the call's argument, so first the old value of x is saved on the stack. After the call, do_return restores x to its previous value.

```
read x;
Cd := hd hd x;      (* Expr. to be evaluated *)
B := tl hd x;       (* Body of fcn. definition *)
Val := tl x;        (* Init. val. of simulated x *)
Stk := 'nil;        (* Computation stack *)
while Cd do STEP;
x := hd Stk;        (* Answer on stack top *)
write x
```

7.2 Other computational models

Selective updating

Very similar results can be obtained for a more powerful version $\mathtt{I}^{su}$ of the language which allows selective up-

Syntactic category:		*Informal syntax:*	*Concrete syntax:*
P : Program	::=	E whererec f(x) = E′	(E.E′)
E : Expression	::=	x $\mid$ ′nil $\mid$ hd E $\mid$ tl E	x $\mid$ nil $\mid$ (hd E) $\mid$ (tl E)
	$\mid$	cons(E, E′)	(cons E E′)
	$\mid$	if E then E′ else E″	(if E E′ E″)
	$\mid$	f(E)	(call E)

Figure 6: F*-program syntax: informal and concrete*

$[\![\texttt{p}]\!] : D \to D \cup \{\bot\}$, $\mathcal{E}[\![\texttt{E}]\!] : D \to D \to D \cup \{\bot\}$	$t_{\texttt{p}} : D \to \mathcal{N} \cup \{\infty\}$, $te_{\texttt{E}} : D \times D \to \mathcal{N} \cup \{\infty\}$
$[\![\texttt{E whererec f(x) = B}]\!]\,\texttt{d} = \mathcal{E}[\![\texttt{E}]\!]\,\texttt{d}\,\texttt{B}$	$t'_{\texttt{Ewhererecf(x)=B}}(\texttt{d}) = 1 + te_{\texttt{E}}(\texttt{d},\texttt{B})$
$\mathcal{E}[\![\texttt{x}]\!]\,\texttt{d}\,\texttt{B} = \texttt{d}$	$te_{\texttt{x}}(\texttt{d},\texttt{B}) = 1$
$\mathcal{E}[\![\texttt{if E then E}' \texttt{ else E}'']\!]\,\texttt{d}\,\texttt{B} = \text{if } \mathcal{E}[\![\texttt{E}]\!]\,\texttt{d}\,\texttt{B} \text{ then}$	$te_{\texttt{ifEthenE}'\ \texttt{elseE}''}(\texttt{d},\texttt{B}) = 1 + te_{\texttt{E}}(\texttt{d},\texttt{B}) +$
$\ldots$	$\ldots$
$\mathcal{E}[\![\texttt{tl E}]\!]\,\texttt{d}\,\texttt{B} = (\textit{if } \texttt{(t.u)} = \mathcal{E}[\![\texttt{E}]\!]\,\texttt{d}\,\texttt{B}\,\textit{then}\,\texttt{u}\,\textit{else}\,\texttt{nil})$	$te_{\texttt{tlE}}(\texttt{d},\texttt{B}) = 1 + te_{\texttt{E}}(\texttt{d},\texttt{B})$
$\mathcal{E}[\![\texttt{f(E)}]\!]\,\texttt{d}\,\texttt{B} = \mathcal{E}[\![\texttt{B}]\!]\,\texttt{d}'\,\texttt{B} \text{ where } \texttt{d}' = \mathcal{E}[\![\texttt{E}]\!]\,\texttt{d}\,\texttt{B}$	$te_{\texttt{f(E)}}(\texttt{d},\texttt{B}) = 1 + te_{\texttt{E}}(\texttt{d},\texttt{B}) + te_{\texttt{B}}(\texttt{d}',\texttt{B})$

Figure 7: Meanings and running times of F*-programs*

dating of data structures. Its semantics is based on a Lisp-like execution model.

An example of selective updating is the assignment x.hd.tl := E. Informally, the value of x must be a pointer to a node m containing (s.t) where s is a pointer to a node n containing (u.v). The effect is to evaluate E yielding, say, w, and then to *replace* the tail component v of n by w, so that s now points to the same node n, modified to contain (u.w).

This facility is available in Schönhage's SMMs (Storage Modification Machines [13]), and is easily programmed in Knuth and Tarjan's pointer machines. Perhaps surprisingly, it has been well-known for many years in the Lisp programming language under the names RPLACA/RPLACD and in Scheme as well, as setcar/setcdr.

Jones and Muchnick did static analysis of programs with just this form of selective updating in 1981 [7].

Invariance of our results

Slight extensions of the techniques just seen show that $\texttt{I}^{su}$ and $\texttt{F}^{su}$ can "efficiently" interpret one another. Further, both models can efficiently interpret and be interpreted by Storage Modification Machines, provided one uses the "more realistic and precise measure" of computation time mentioned by Schönhage [13]. These are in turn "efficiently" equivalent (given a similarly fair accounting of instruction costs) to Knuth and Tarjan's pointer machines, and to successor RAMs [13,14,11].

Consequently, our three main results (constant hierarchy, existence of complete problems, efficient solutions of the Recursion Theorem) hold for all these models, and all the models define the same class $\mathbf{LIN}^{su}$ of problems solvable in deterministic linear time.

8 On time measures

8.1 Implementing languages I and $\texttt{I}^{su}$

I-program execution times assume standard Lisp implementation technology, where execution proceeds by a sequence of updates and accesses to a "store". A store is a dag (ordered directed acyclic graph) with atoms as leaf labels and with internal "cons" nodes labeled ".". Any expression value is (a pointer to) a dag node. Evaluation of hd E or tl E gives (a pointer to) the left or right son of the node which is the value of E, and nil if that node is a leaf. Finally, E1.E2 evaluates to a new node, whose left and right sons are the values of E1 and E2. Program output is obtained by unfolding the dag to obtain a tree in D.

That this technique is semantically faithful to the tree rewriting interpretation of Figure 2 has been formally proven in [2].

Essentially the same idea is used for $\texttt{I}^{su}$-programs, except that the storage structure becomes a graph, possibly cyclic. Consider a command such as A.hd := E where A denotes a node n containing, for example, (u.v). Expression E is evaluated to, say, w, and then the content of node n is changed to (u.w). Note that a new node is *not* created, so any pointers to n now refer to its updated contents.

Clearly both I and $\texttt{I}^{su}$ can be implemented on a unit-cost successor RAM in times proportional to those of Figure 2.

8.2 Numbers of variables, atoms, etc.

The restriction to one variable, one atom, one recursively defined function, etc. may seem artificial, and indeed the invariance results of Section 7 for successor RAMS and pointer machines require accounting for the number of registers and variables used, and the values of program constants.

However, the limitation on the number of atoms or values of constants seems essential, since its lack is the sole reason for the possibility of arbitrarily large constant speedups in the Turing machine model. This is a result of questionable value, as it is proven by an encoding technique that is totally useless for practical programming.

Code	Stack	Value	Body	⇒	Code	Stack	Value	Body
`('call E).Cd`				⇒	`E.'do_call.Cd`			
`'do_call.Cd`	`w.s`	`v`	`B`	⇒	`B.'do_return.v.Cd`	`s`	`w`	`B`
`'do_return.v.Cd`	`u.s`	`w`		⇒	`Cd`	`u.s`	`v`	

Figure 8: `STEP` *macro rules for function calls.*

8.3 Fair charges for program variable access

Supposing that we extend language `I` to allow variables $\mathtt{x}_1$, $\mathtt{x}_2$,..., how should the time function $t_{\mathtt{p}}(\mathtt{d})$ be modified to obtain a realistic measure of computation time?

If at the one extreme we charge only 1 time unit for any reference to $\mathtt{x}_i$, regardless of i, it appears to be impossible to construct an "efficient" universal program `u`. (The reasoning is that the current values of all the interpreted program's variables would have to be placed in one of `u`'s data structures. This would grow in size with the number of the interpreted program's variables, so the time to access $\mathtt{x}_i$'s value would depend on i.)

At another extreme, one could charge i time units for a reference to $\mathtt{x}_i$. This would model the common technique of representing program `p`'s store as a linear list, but it leads to another unpleasant consequence: one could execute any program with sufficiently frequent references to sufficiently many variables *faster by running it interpretively* than by running it directly.

This could be done by constructing a (fixed) universal program `u'` in which any input program `p`'s store is represented as a binary tree, allowing fast access or modification to $\mathtt{x}_i$. The time `u'` saves by using its binary tree (time $O(\log i)$) as opposed to program `p`'s time $O(i)$ will, for sufficiently large programs, offset the interpretation overhead.

The conclusion is that a charge of $O(\log i)$ time units to access or modify $\mathtt{x}_i$ seems fair for a multivariable version of language `I`, since with this measure an efficient universal program exists, and most large programs cannot be run faster by executing them interpretively.

8.4 Unlimited numbers of program variables

The restrictions on the numbers of variables or registers may be lifted for stronger languages with selective updating such as SMMs or $\mathtt{I}^{su}$, at least for running times $O(n \log n)$ or larger. The key to all the previous results was the construction of an efficient universal program, so it suffices to see how this can be done without limitation to a priori bounded numbers of variables.

The technique is for the universal program to preprocess its program input before interpreting, translating it into a directed graph in which variable references are linked directly to their storage cells. Preprocessing can be done in time $O(n\log(n))$, e.g. by using a binary tree as symbol table to bind variables to their storage addresses. This time factor is added on before efficient interpretation begins, after which the methods used in earlier sections can be applied. The direct pointer bindings allow constant access time.

It is unclear whether an analogous result holds for languages without selective updating.

8.5 Is the time measure realistic?

Some readers may feel our model to be unrealistic in that we count only one time step for any access through a pointer, however large, rather than time proportional to the number of bits occupied by its pointer address.

A first point is that in spite of the argument that one should "charge" time proportional to the address length for access to a dag or graph node, this is not the way people think or count time when they program. Computer memories are carefully designed to make pointer access a very fast operation, so users rarely need to be conscious of address length in order to make a program run fast enough. Further, hardware is fixed: word sizes or memory capacities cannot practically be increased during execution (the "trick" used to get constant Turing machine speedups).

For the time being, we just note that our most novel results concern linear time algorithms, which simply do not run long enough to fill sufficiently many memory cells to create long addresses or values.

Costs in addressing both data and program code

The devil's advocate: we just argued that nonconstant measures have very little effect on linear time computations. Still, in $O(n)$ steps one may allocate $O(n)$ nodes with addresses requiring up to $\log(n)$ bits — a function depending on n. Does this rule out the possibility of an "efficient" universal program with overhead independent of the interpreted program's size, thus *rendering invalid all the proofs seen above*?

In our opinion it does not, for the following reason. Suppose one assumes that all memory accesses take time proportional to the length of the address. Then the time charged to execute a program instruction should account not only for the time to access its data and to do the operation ordered (e.g. to add two numbers), but also for *the time to locate the instruction itself.* This factor is proportional to the length of the instruction's address in memory.

Given such an accounting practice, a universal program can store the program to be interpreted in odd memory locations, and can represent program memory cell *loc* in the interpreter's memory cell $2 \cdot loc$. Under logarithmic cost this adds only 1 to each interpreted instruction fetch or memory cell access, so the total interpretation time will still be bounded by a constant times the interpreted program's running time.

9 Conclusions and acknowledgements

It is common wisdom in complexity theory that linear changes in time bounds have no effect at all on computation power. On the other hand such changes *are* significant in daily practice. In particular, automatic program transformation by *partial evaluation* nearly always gives linear speedups [8,9]. Our main result thus brings theory closer to practice, and reduces an implicit tension between complexity theory and partial evaluation.

Several new results have been proven for a simple language with one-variable, one-atom programs manipulating lists: the existence of a hierarchy based on constant time multiples, in contrast to the situation for Turing machines; the existence of complete problems for a nondeterministic version of the language; and surprisingly efficient versions of the programs whose existence is guaranteed by Kleene's second recursion theorem. The language is powerful enough for linear time to be interesting, and results are robust with respect to machine model.

Acknowledgements Thanks are given to Henk Barendregt, John Case, Yuri Gurevich, Joel Seiferas, Dieter Spreen, and the programming language research groups at the Universities of Copenhagen and Aarhus.

References

[1] Torben Amtoft, Thomas Nikolajsen, Jesper Larsson Träff, Neil D. Jones: Experiments with Implementations of two Theoretical Constructions. Logic at Botik, Lecture Notes in Computer Science, Springer-Verlag, vol. 363, pp. 119-133, 1989.

[2] H. Barendregt et. al. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, P. C. Treleaven (eds.): *PARLE—Parallel ARchitectures and Languages Europe* (Lecture Notes in Computer Science, vol. 259), 1989.

[3] Manuel Blum: A Machine-Independent Theory of the Complexity of Recursive Functions. Journal of the Association for Computing Machinery, vol. 14, no. 2, April 1967, pp. 322-336.

[4] J. Royer and J. Case: Intensional subrecursion and complexity theory. Research notes in theoretical science. Pitman Press 1988.

[5] Yuri Gurevich, Saharon Shelah: Nearly linear time. Logic at Botik, Lecture Notes in Computer Science,Springer-Verlag, vol. 363, pp. 108-118, 1989.

[6] J. E. Hopcroft, J. D. Ullman: Introduction to automata theory, languages, and computation. Addison-Wesley, 1979.

[7] N.D. Jones, S. Muchnick:Flow analysis and optimization of Lisp-like structures. *Program Flow Analysis*, eds. N.D. Jones, S. Muchnick, pp. 102-131. Prentice-Hall, New Jersey, 1981.

[8] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[9] N.D. Jones, C. Gomard, P. Sestoft: Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993.

[10] Stephen Cole Kleene: Introduction to Metamathematics. North-Holland, 1952.

[11] Donald Knuth: Fundamental algorithhms. Addison-Wesley, 1968.

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

[13] A. Schönhage: Storage modification machines. SIAM Journal of Computing vol. 9, pp. 492-508, 1980.

[14] Robert Tarjan: A class of algorithms which require non-linear time to maintain disjoint sets. Journal of Computer and System Sciences vol. 18, pp. 110-127, 1979.