Computational Complexity via Programming Languages: Constant Factors Do Matter

Amir M. Ben-Amram, Neil D. Jones*

May 31, 2000

Abstract

The purpose of this work is to promote a programming-language approach to studying computability and complexity, with an emphasis on time complexity. The essence of the approach is: a programming language, with semantics and complexity measure, can serve as a computational model that has several advantages over the currently popular models and in particular the Turing machine. An obvious advantage is a stronger relevance to the practice of programming. In this paper we demonstrate other advantages: certain *proofs and constructions* that are hard to do precisely and clearly with Turing machines become clearer and easier in our approach, and sometimes lead to finer results. In particular, we prove several *time hierarchy theorems*, for deterministic and non-deterministic time complexity which show that, in contrast with Turing machines, constant factors *do* matter in this framework. This feature, too, brings the theory closer to practical considerations.

The above result suggests that this framework may be appropriate for studying low complexity classes, such as linear time. As an example we give a problem complete for *non-deterministic* linear time under deterministic linear-time reductions. Finally, we consider some extensions and modifications of our programming language and their effect on time complexity results.

1 Introduction

Complexity theory has many faces. Its fields of application range from the highly abstract axiomatic complexity of Blum to the very concrete com-

*Both authors's work was supported in part by DART, a project under the Danish Research Councils. Work of the first author was partly supported by a grant from the Nachemsohn Memorial Fund, Denmark. Authors' e-mail: {amirben,neil}@diku.dk

plexity of Boolean circuits. Yet its foremost application is probably to the efficiency of software. We wish the theory to tell us what we can accomplish with computer programs when computational resources, such as computing time, are limited (or just costly). This view stands in some tension with the predominant kind of computational models in complexity theory: abstract machines. In models such as the Turing machine, the random access machines and others, the answer to the question "what constitutes an algorithm" is *algorithm=machine*. Thus, a computational model is a class of "machines," where each machine embodies a single algorithm. We refer to this approach as the *hardware view* of computation.

This paper is part of an effort to promote an approach that we call *the software view*, where the answer to the above question is *algorithm=program*. We claim that the software view bears more than a nominal change to make theory seem more relevant to programmers (though this also has some value). We propose that it is the *programming language*, far more than the hardware, that dictates what a programmer can do, and creates the framework for reasoning about programs. Thus the language is the programmer's computational model.

Important properties of a computational model are power, flexibility and simplicity. Among the popular machine models, the RAM is certainly a powerful model, and allows for very flexible programming, but often the simplicity of the Turing machine causes theoreticians to prefer its usage. We demonstrate that a programming language can be both simple and powerful enough to make analysis, as well as program constructions, encountered in theoretical work manageable. Such constructions frequently involve subtle programming techniques such as metaprogramming — where programs are treated as data objects, to be interpreted or transformed by other programs. Programming such operations on the Turing machine (and other models) is so tedious that most authors resort to sketchy algorithm descriptions.

Using a programming language of the style presented in this paper, we can carry out the constructions out with sufficient programming detail to give a sense of completeness to our proofs. The key to the relative ease of metaprogramming in our language is *structure*. Our programming language has a structured syntax, using nested commands in contrast with the typical machine's transition diagram. Moreover, it operates on *structured data*, a feature which removes the need for hard-to-program "pairing functions" and other encoding schemes that are needed with popular abstract machines.

Another gap between current theoretical practice and real-world programming is the usage of "big O" notation, where the disregarded constant factor is obviously of concern in practical applications. Theoreticians often justify their use of asymptotic notation by the difficulty in exactly figuring out the complexity of a program. This is obviously true when the computational model you use is so hard to program that writing programs fully is not reasonable. Another argument has its theoretical foundation in the Turing machine constant speedup theorem [16, 19], which shows that in Turing machine complexity, constant factors *are meaningless*. However, the proof of the speedup theorem involves changing the machine's tape alphabet — while keeping the time to branch according to the value of a symbol unchanged. This roughly corresponds to enlarging the word size in a computer without increasing the instruction times. While this change can be achieved from time to time by a newer hardware technology, there are definite bounds to this growth, and at any rate this change is not under the control of the *programmer*. Hence, it is intuitively reasonable that a *programmer-oriented* theory should not contain the constant speedup theorem.

Using the programming language described in this paper, we prove a series of hierarchy theorems, similar but less coarse than the hierarchy theorem known for Turing machines. The theorems show that a constant-factor difference in time makes a difference in problem-solving power. The main theorem for deterministic time states that for time-constructible functions $T(n)$, there is a constant b such that more decision problems can be solved in time $bT(n)$ than in time $T(n)$. The corresponding theorem for non-deterministic time is similar but has $bT(n+2)$ instead of $bT(n)$. This difference results from the method of proof, but is not significant for functions T that do not grow too fast.

Sudborough and Zalcberg [35] proved a similar deterministic-time hierarchy theorem for a unit-cost random access machine. In fact, the random access machine enjoys many of the advantages that we claim for our programming model, and in fact it closely reflects the assembly programmer's view of a typical computer. In this sense, the RAM can be easily accommodated into the software approach, albeit in the form of a very low-level language.

We now describe the essential features of the programming language that we use. Our language is a structured imperative language. The ease of programming and metaprogramming is a result of combining the structured programming language with structured data. For simplicity, our language has only one data type: binary trees, known as S-expressions to LISP speakers. This allows for much versatility in programming, for example instead of devious "pairing functions", we can simply use the tree cons operation. We use the set of trees as our input/output domain as well. This is unorthodox in that the natural input/output domain in complexity theory is one of strings over a finite alphabet. However, working with trees instead does not create any essential difficulty, and for running time that is at least linear, it makes no difference in complexity results, because data can be converted from one representation to the other. This transformation is important in order to make comparisons of the power of our languages with traditional computational models. However in this paper we will mostly work within our framework. One of the important benefits that we obtain from having a structured language together with structured data is the ability to interpret programs efficiently, i.e., to program an *efficient universal program.* Such a universal program is a useful tool in proving complexity results, in particular the hierarchy theorems mentioned above. The data type and the programming language are described in Sections 2 and 3, followed by the construction of *efficient universal programs* in Sections 4 and 5 and their use in proofs in Section 6.

Every computational model ever considered brings with it the question of how various modifications affect its power, e.g., Turing machines can have different numbers of tapes, heads, non-linear tapes etc. In the realm of programming languages, features that may be considered include recursive function calls, `goto`'s, and different operations on storage, such as selective update of aggregates. Such modifications and their effects on a language's complexity-related properties are interesting to study and perhaps more clearly relevant to programming than modifications of Turing machines. In Section 7 we consider extensions of our base language, with stronger control over storage, extensions which possibly make the language stronger with respect to time complexity. We cannot resolve this question, but we are able to show that results we have obtained using the base language can be strengthened using the extensions. Changes of programming style (such as functional versus imperative) are briefly considered later in the paper.

The hierarchy theorems, which even stratify linear time into an infinite hierarchy, suggest that our language may allow for studying this small complexity class. Section 8 gives a simple problem that is complete for *non-deterministic linear time*, a concept analogous to the well-known one of NP completeness.

This paper is part of a larger effort to base studies in computability and complexity on programming languages, more or less similar to those presented in this paper. This work started with Neil Jones' 1993 STOC paper [18], and has already resulted in a textbook [19]. The basic definitions in this paper are drawn from these references. Some work by other authors which seems related to the major themes of this paper is reviewed in Sections 9 and 10.

2 Structured data: binary trees

While being simple, our programming language is quite powerful, and this owes a lot to the fact that it is designed to express computations on *structured data*. More precisely, computations over a domain $\mathbb{D}_A$ of binary trees: this is the natural domain for LISP programs, so a reader with knowledge of LISP will find the following definitions familiar. Trees are built up from a fixed finite set A of so-called *atoms* by finitely many applications of the pairing operation ".". Thus a value in $\mathbb{D}_A$ is a binary tree with atoms as leaf labels. It can be written down in 'fully parenthesised form', e.g.:

$$((\mathtt{a}.((\mathtt{b}.\mathtt{nil}).\mathtt{c})).\mathtt{nil})$$

We mostly use only one atom `nil` so $A = \{\mathtt{nil}\}$. When the set A is clear from the context $\mathbb{D}_A$ will be written $\mathbb{D}$. As a convention, values in $\mathbb{D}_A$ are written in typewriter face (as above). The letter `d` is usually used to designate an element of $\mathbb{D}$. The *size* of `d` is defined inductively: $|\mathtt{d}| = 1$ if `d` is an atom, and $|\mathtt{d}_1 \,.\, \mathtt{d}_2| = 1 + |\mathtt{d}_1| + |\mathtt{d}_2|$. Thus $|\,((\mathtt{a}.((\mathtt{b}.\mathtt{nil}).\mathtt{c})).\mathtt{nil})\,| = 9$. Note that $|\mathtt{d}|$ is always an odd number.

In writing values from $\mathbb{D}$, outer parentheses will sometimes be omitted for brevity, e.g. "`b.c`". pairing is right-associative, so `a.b.c`=`a.(b.c)`. We also use LISP's *list notation*, where the value

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

is written as

$$(\ \underline{\mathtt{d}_1}\ \cdots\ \underline{\mathtt{d}_n}\ )$$

where $\underline{\mathtt{d}_i}$ is either $\mathtt{d}_i$ or a list notation for $\mathtt{d}_i$. For example, the tree

$$((\mathtt{a}.(\mathtt{b}.\mathtt{nil})).(\mathtt{c}.\mathtt{nil}))$$

can be written as `((a b) c)`.

2.1 A representation for integers

Some of the programs in this paper manipulate integer values. To this end, we represent the number $n \geq 0$ as $\mathtt{nil}^n = \mathtt{nil}.\mathtt{nil}.\cdots.\mathtt{nil}$, with $n+1$ `nil`'s (hence $\mathtt{nil}^0 = \mathtt{nil}$). Every reference to "a variable holding an integer" or "an integer value" in the sequel should be understood as referring to a value of the above form.

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	$\in$	$\mathbb{D}$	

Figure 1: Command and expression syntax: informal and concrete

3 Structured programs: the language I

In this section we introduce the programming language I, which is our model of computation. To model computability as well as complexity, we endow programs with both semantics and timing. We also discuss several kinds of extensions of the language that will be useful in the sequel, and give a few programming examples.

In general, with every programming language L we associate two sets, L-data, which is the domain on which functions can be computed by L programs, and L-programs, the set of well-formed programs. For language I, we have I-data= $\mathbb{D}$, while program structure is defined next.

3.1 Syntax of language I

Figure 1 gives I expression and command syntax. The "concrete syntax" shows how programs are represented as data values in $\mathbb{D}_A$ for atom set $A = \{:=, ;, \texttt{while}, \ldots, \texttt{cons}\}$. Note that the syntax presented only allows commands to manipulate a single variable, X. We also restrict the set of data values that can appear in a program to $\mathbb{D}$, trees with the single atom `nil`. We refer to the language as defined here as *the base language*. Usage of additional atoms and additional variables is discussed later on.

Input and output in language I are values in $\mathbb{D} = \mathbb{D}_{\{\texttt{nil}\}}$. In order to allow I programs to be read and written by I programs we can adopt an encoding of the extra atoms mentioned above as distinct values in $\mathbb{D}_{\{\texttt{nil}\}}$. The choice of the particular values is not important.

In writing programs informally, we use the informal syntax shown in Figure 1, augmented with the use of indentation and curly braces to clarify program structure. We remark that we do not distinguish in notation between the objects "program" and "the code of that program" (a tree). No confusion will arise.

3.2 Semantics and time usage

The semantics of the language are just what you expect, so we omit a formal definition, which can be found in [19]. The result of running program `p` on input `d` is denoted by $[\![\mathtt{p}]\!]\mathtt{d}$ and is a value in $\mathbb{D}$ or is undefined (for non-terminating programs). Running time is defined by a unit-cost criterion. Thus accessing a variable or a constant, assignment, testing for `nil` and each of the tree operators cost one time unit. We charge no time for I/O as our program is not intended to model I/O operations. We denote the running time of program `p` on input `d`, in general, by $time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d})$ where `L` is the language in which `p` is written. For language `I` which is the basic language for this paper we often omit the superscript. We say that program `p` runs *within time* f if f is a function such that $time_{\mathtt{p}}(\mathtt{d}) \leq f(|\mathtt{d}|)$ for all $\mathtt{d} \in \mathbb{D}$.

Note that a command `X := cons X X` binds `X` to a new tree with size more than twice the size of the previous value of `X`. Therefore it may seem at first sight unreasonable to assume unit cost for tree operations. In fact, this cost model is appropriate when we consider a *data-sharing implementation* as in LISP and newer functional languages. Here is a brief summary of this technique:

1. At any time the storage is a directed acyclic graph (DAG). It has only one sink, labelled `nil`, and every internal node has two children labelled `hd` and `tl`.

2. The value of `X` is always a pointer to a node in the DAG.

3. Evaluating expression `E` may add new nodes to the DAG (as results of `cons` operations), and yields as value a pointer to a node in the (possibly modified) DAG.

In this method, every basic operation of `I` is indeed implemented in constant time (on a unit-cost random access machine).

3.3 Nondeterministic programs and acceptance

A special kind of program in which we are interested is the *acceptor*, whose output is either `nil` (signifying non-acceptance) or non-`nil` (signifying acceptance). It will sometimes be convenient to use the terms "acceptance" or "rejection" instead of referring to the output values.

It is mainly in conjunction with acceptors that we will be interested in *nondeterministic programs*. The power of nondeterminism is added to `I` by means of the `choose` command. Its effect is to choose one of two commands nondeterministically. *Informal syntax*: `choose C1 | C2`. *Concrete syntax*: `(choose C1 C2)`

A nondeterministic acceptor program is said to accept its input if there *exists* a sequence of nondeterministic choices that leads to acceptance. In this case, we define the time complexity of the program to be the running time of the shortest accepting computation (with the `choose` instruction accounting for one time unit). If there is no such computation, the running time is undefined. Here are some formal notations:

Definition 3.1 *Suppose we are given a programming language* `L` *with corresponding sets* `L`-programs *and* `L`-data *(in particular we consider the language* `I` *with* `I`-data $= I\!D$*).*

1. *For any* deterministic *program, we define*

$$Acc^{\mathtt{L}}(\mathtt{p}) = \{\mathtt{d} \in \mathtt{L}{-}\mathrm{data} \mid \mathtt{p} \text{ accepts } \mathtt{d}\}$$

2. *For any* nondeterministic *program, we define*

$$Acc^{\mathtt{L}}_{\exists}(\mathtt{p}) = \{\mathtt{d} \in \mathtt{L}{-}\mathrm{data} \mid \mathtt{p} \text{ accepts } \mathtt{d}\}$$

In all these notations we omit the superscript `L` *when it can be understood from context.*

3.4 Syntactic sugar

By "syntactic sugaring", we mean extending the language by constructs which can be easily replaced with constructs of the base language (examples follow). Hence there is no need to add them to the concrete syntax, or to consider them in reasoning about programs. Their whole purpose is to enhance the readability of programs.

Some simple "sweeteners" are:

1. "E1.E2" is a shorthand for "cons E1 E2".

2. Connectives and, or, not will be used in while and if statements, with the expected meaning (e.g., the conditional E1 and E2 is satisfied if and only if both E1 and E2 are non-nil).

3. Sometimes we treat tree values as integers. This will only be done in a context where it is understood that X is of the form $\mathtt{nil}^i$ and can be interpreted as representing an integer as in Section 2.1. Note that such a "number" can be incremented, decremented and tested for zero, all in constant time.

3.5 Macros

The ability to define subprograms and to call them greatly enhances the ease of programming in any language, and it is therefore important that these can be accommodated easily.

In fact, as long as we do not need the subprograms to be *recursive*, we can do with simple *macro programming.* A macro is a command which is defined separately and given a name; referring to this name is tantamount to including the macro code in the program. The simplest case is a macro with no argument, which is included simply by quoting its name. This is nothing more than a shorthand notation. We can also use a macro as a "function" which accepts an argument and returns a result. Such a function is "called" by the command X := f(Exp), a shorthand for

```
X := Exp; code of f
```

Thus the "return value" of the "function" f is simply the value that it leaves in X. We can also use multiple-argument notation, such as: X := f(E1,E2,E3). The formal meaning of this construct is

```
X := E1.E2.E3; code of f
```

We emphasize again that macro programming does not involve any change of the actual language.

3.6 Extensions of the language

In this subsection we consider extensions more essential than the syntactic sugaring above. Extended programs can be translated into I programs, but at the price of a slowdown. We will be interested in distinguishing a *constant-factor slowdown* from a slowdown that is not bounded by any constant (an example to the latter is a translation that squares the running time). Moreover, we distinguish constant-factor slowdowns that are *program-dependent* from *program-independent* ones (where the same constant bounds the slowdown for all programs).

3.6.1 More atoms and variables

First, we show that I-programs can simulate programs with many variables or atoms, with at worst a constant slowdown depending on the numbers of variables and atoms (the reason that a slowdown is incurred, is that we do not change our time measures — for instance every variable access still accounts for one time unit. This approach will be maintained through all the language changes we discuss).

More atoms. Assume for example that a given program uses atoms `0` and `1` in addition to `nil`, so $\mathbb{D} = \mathbb{D}_{\{\texttt{nil}\}}$ is extended to $\mathbb{D}' = \mathbb{D}_{\{\texttt{nil},0,1\}}$. The extended syntax and semantics are the same except that the atom set is larger, and so we need a means of testing for the identity of an atom. This may be done by the inclusion of a predicate `atom=?` such that `atom=? E1 E2` evaluates to `true` if the expressions `E1` and `E2` evaluate to one and the same atom.

A list structure $\mathtt{d} \in \mathbb{D}'$ may be encoded as $\overline{\mathtt{d}} \in \mathbb{D}$ as follows.

$$\begin{array}{rcl} \overline{\mathtt{nil}} & = & \mathtt{nil} \\ \overline{\mathtt{0}} & = & \mathtt{nil.nil} \\ \overline{\mathtt{1}} & = & \mathtt{nil.(nil.nil)} \\ \overline{\mathtt{d1.d2}} & = & ((\mathtt{nil.nil})\ .\ (\overline{\mathtt{d1}}\ .\ \overline{\mathtt{d2}})) \end{array}$$

Given this representation, $|\overline{\mathtt{d}}| \leq 5 \cdot |\mathtt{d}|$. The operations `hd`, `tl` and `cons` on `d` can all be simulated on $\overline{\mathtt{d}}$ in constant time, and similarly for the test `atom=? d1 d2`.

In general, we obtain an expansion of the program code by a constant factor that depends on the number of added atoms. The running time of the translated program is also multiplied by a similar constant.

Note that programs in the extended language can recognize the additional atoms in the input too. The one-variable program obtained by the translation needs the input to be encoded using the same scheme.

More variables. Consider a program with three variables `Cd`, `Stk`, `Val` (an example that we later use). One can convert it to an equivalent one-

```
Y := nil.nil;
while X do
 if (hd X) then
  X := (hd hd X).((tl hd X).(tl X))
 else
  Y := nil.(nil.Y);
  X := tl X
```

Figure 2: Procedure to compute the size of a tree.

variable program by "packing" the three into one: X = Cd.(Stk.Val). In this way, a command like Stk:=Val.Stk can be realized as follows:

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

As above, this conversion (generalized to any number of variables) entails code expansion and slowdown by a constant factor that depends only on the number of variables.

Example. Figure 2 shows a program to compute the size of a tree, using two variables: X, as usual, holds the input, and Y is used as a counter for computing the size. We leave it to the reader to verify that the program is correct and runs in time linear in the input size. When we convert the program to use a single variable, the time remains linear.

3.6.2 Extensions with program-dependent slowdown

Suppose we break uniform boundedness and extend our language with an infinite supply of atomic symbols; i.e., we let $\mathbb{D} = \mathbb{D}_A$ where A is infinite. Any single program can only refer in its code to a finite number of those, but there is no bound on this number. Similarly, we can allow for an arbitrary number of variables in a program (both extensions are present in the LISP language). 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 as well as on its input.* If we restrict the input to $\mathbb{D}_{\texttt{nil}}$ (also for the extended language), we obtain a slowdown that only depends on the new atoms and additional variables appearing in the *program code.*

4 Efficient universal programs

A *universal program* is the theoretician's name for a *self-interpreter*: a program in language L that interprets L-programs. Universal programs are one of the foundations of computability (and often connected with the work of Turing, although in his approach they are of course called universal *machines*). In practical computing, interpreters are ubiquitous, and their efficiency is often important. In complexity theory, interpreters are used in various constructions, a typical example being diagonalization proofs. Here, too, efficiency matters. One of the advantages of our programming language is that it has an "efficient" self-interpreter in a precise sense defined below. For the existence of a self-interpreter it is required that the programs be represented as data objects, in other words have a concrete syntax representation.

Definition 4.1 *Suppose we are given a programming language* L *with corresponding sets* L-programs *and* L-data*, where* L-programs$\subset$L-data *(in particular we consider language* I *with* I-data $= \mathbb{D}$*).* L*-program* u *is an* L self-interpreter *(or* universal program*) if for every* d $\in$ L-data *and* L *program* p

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

Note that this equality is of partial values and includes the case "undefined".

Definition 4.2 *An* L *self-interpreter* u *is* efficient *if there are constants* a *and* b *such that for every* d $\in$ L-data *and* L *program* p

$$time_{\mathtt{u}}(\mathtt{p}\ .\ \mathtt{d}) \leq a \cdot time_{\mathtt{p}}(\mathtt{d}) + b \cdot |\mathtt{p}|$$

Note that the constant a is quantified *before* p, so the multiplicative slowdown caused by an "efficient" interpreter is independent of p. The linear additive term would typically account for preprocessing of the program text. We remark that in our applications of efficient interpretation, we could do with a more relaxed definition where the preprocessing cost can be more than linear in $|\mathtt{p}|$. However, since we are able to achieve linear cost with the languages we consider we found it convenient to use the restrictive definition.

4.1 A universal program

We now present an efficient self-interpreter for I. Recall that an I program is given in a concrete syntax that includes some extra atoms (`if`, `:=`, etc).

We present our self-interpreter using these atoms, and some additional ones that will appear below. It should be understood, however, that everything is actually encoded in $\mathbb{D}_{\texttt{nil}}$, so that our interpreter is an `I` program. We also grant ourselves the convenience of using several named variables (see Section 3.6.1).

The interpreter is given by program `u` below, where STEP, the main loop's body, will be described next. The program uses two stacks, `Cd` and `Stk`, that are represented in the natural way as lists.

```
read X;               (* X = (p.d) *)
Cd := (hd X).nil;     (* Code stack *)
X := tl X;            (* Initial value of X *)
Stk := nil;           (* Computation stack *)
while Cd do STEP;
write X
```

The purpose of the STEP macro is to remove the head of the code stack, and according to its value perform an operation that may result in changing the current value `X`, the computation stack and the code stack itself. For clarity, we present this part of the program as a table of rewrite rules rather than a series of `if`'s and assignments (below we give an example of `I` code corresponding to one of the rules). Blank entries in Table 3 correspond to values that are neither referenced nor changed in a rule; the column "+" is a special mark column, which is used at the moment only for purposes of analysis. We remark also that the last rule (`nil` $\Rightarrow$ `nil`) is never used by the interpreter but is included for future applications of this macro.

Example. Here is the `I` code for the rules to handle `while` commands (we use the binary operator `=` for `atom=?`):

```
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 ...
```

Cd	Stk	X	*new*: 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*

Lemma 4.3 u *is a self-interpreter.*

PROOF: The claim means that the result of running u on (p.d) equals $[\![p]\!]$d. This is easily verified by routine induction. □

We remark that the correctness of interpreter is only meaningful for input (p.d) where p is an I program. If desired, u could be augmented to verify that the head of the input actually represents a program (in other words, that p is well formed). This is not hard to program, but is not necessary for our purposes.

Lemma 4.4 *During execution of program* u *on input* p.d *the number of invocations of rules marked by* + *in Table 3 equals* $\mathit{time}_{\mathrm{p}}(\mathrm{d})$.

Proof: Each one of these rules is invoked once in correspondence with one unit-cost operation of the interpreted program. $\Box$

Theorem 4.5 u *is an efficient self-interpreter.*

Proof: Correctness has already been stated. As to time, first note that the entire STEP macro is a fixed piece of non-iterative code: the commands to find the matching rule in the table and realize its effect take time bounded by a constant independent of the interpreted program p. The code outside the main loop also takes constant time, so the running time of u satisfies

$$time_{\mathtt{u}}(\mathtt{p} \mathbin{.} \mathtt{d}) \leq c \cdot S(\mathtt{p}, \mathtt{d}) + b$$

for constants b, c where $S(\mathtt{p}, \mathtt{d})$ is the number of invocations of STEP when program p is interpreted with input d.

We now evaluate $S(\mathtt{p}, \mathtt{d})$. The previous lemma gives the number of times that rules marked by + in the table are applied. Note that every unmarked rule extends the code stack. Only marked rules pop the stack, and at most two elements are popped at once. Since the program stops with an empty stack, we can bound the number of applications of unmarked rules by twice the number of applications of marked rules, hence by $2time_{\mathtt{p}}(\mathtt{d})$.

We conclude that $S(\mathtt{p}, \mathtt{d}) \leq 3time_{\mathtt{p}}(\mathtt{d})$, hence letting $a = 3c$ we have

$$time_{\mathtt{u}}(\mathtt{p} \mathbin{.} \mathtt{d}) \leq a \cdot time_{\mathtt{p}}(\mathtt{d}) + b \leq a \cdot time_{\mathtt{p}}(\mathtt{d}) + b \cdot |\mathtt{p}|.$$

$\Box$

5 More universal programs

In this section we present some variations on the efficient universal program of Section 4: a timed universal program and several sorts of universal programs for the nondeterministic model. These programs are all used in the next section.

5.1 A timed universal program

Definition 5.1 I-*program* tu *is an* efficient timed interpreter *if there are constants k, l such that for every* I *program* p, *value* $\mathtt{d} \in I\!\!D$ *and* $n \geq 1$*:*

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

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

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

Thus, a timed interpreter differs from an ordinary one in that it limits the running time of the interpreted program according to an input value. We construct an efficient timed interpreter `tu`, based on the self-interpreter `u`. We add a new variable, `Cntr`, to "count down" the running time. We modify the STEP macro to decrease `Cntr` at every rule that is marked by a `+`. The main interpretation loop is modified to

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

and followed by a test to determine the output:

```
 if Cd then
  X := nil
 else
  X := X.nil;
 write X
```

Theorem 5.2 `tu` *is an efficient timed interpreter.*

Proof: Verifying that `tu` fulfills the first two clauses of the definition (hence that its output is correct) is straight-forward given Lemma 4.4. As for running time, since `tu` adds only a constant amount of time to each iteration as well as to the non-iterative part of the program, its running time can still be bounded by $c \cdot S(\mathtt{p}, \mathtt{d}, n)$, where c is a constant and $S(\mathtt{p}, \mathtt{d}, n)$ the number of STEP's performed for input $\mathtt{p.d.nil}^n$.

To evaluate $S(\mathtt{p}, \mathtt{d}, n)$ we distinguish two cases. *Case* 1: the program runs to completion. Then the reasoning in the proof of the previous theorem applies and we have

$$S(\mathtt{p}, \mathtt{d}, n) \leq 3 time_{\mathtt{p}}(\mathtt{d})\,. \qquad (1)$$

Case 2: the interpreter stops after having decreased the counter n times. This makes a change in the analysis, because the run may be terminated when the code stack is not empty. Thus, the number N^+ of stack push operations cannot be bounded by the number N^- of pop operations: instead, we have

$$N^+ = N^- + R$$

where R is the number of elements that remain in the stack when the program stops. We claim that $R \leq |\mathtt{p}|$. Briefly, this holds because the code stack contains only fragments of the program, and no fragment appears twice[1].

[1] We only count entire program fragments that are pushed unto the stack.

We conclude that the number of push operations to each of these stacks can surpass the number of pop operations by at most $|\mathtt{p}|$. The number of "popping" iterations is now simply bounded by n (since it is for these rules that the counter is decreased). Continuing the reasoning as in the proof for u we have

$$S(\mathtt{p}, \mathtt{d}, n) \leq 3n + |\mathtt{p}|\,. \tag{2}$$

Combining Equations (1) and (2) we obtain

$$\mathit{time}_{\mathtt{tu}}(\mathtt{p}\,.\,\mathtt{d}\,.\,\mathtt{nil}^n) \leq k \cdot \min(\mathit{time}_{\mathtt{p}}(\mathtt{d}), n) + l \cdot |\mathtt{p}|$$

for appropriate constants k, l. □

5.2 Nondeterministic universal programs

We assume our nondeterministic programs to be acceptors. We present three interpreters for nondeterministic programs:

5.2.1 Efficient universal program nu

This is a self-interpreter for the non-deterministic language. It is obtained from the deterministic program u by adding to the STEP macro a rule that implements choose by choose.

5.2.2 Efficient timed universal program tnu

This is obtained by the same modification, applied to tu (with the choose rule marked for timing).

Note that we defined $\mathit{time}_{\mathtt{p}}(\mathtt{d})$ for non-deterministic p by the shortest accepting computation. But tnu, when run on $\mathtt{p.d.nil}^n$, halts within the prescribed time on every computation path.

5.2.3 Deterministic timed universal program dtnu

This program satisfies the following specification: there is a constant k such that for every (nondeterministic) I-program p and values $\mathtt{d} \in D, n \geq 1$:

(1) $\mathit{time}_{\mathtt{dtnu}}(\mathtt{p}\,.\,\mathtt{d}.\,\mathtt{nil}^n) \leq k \cdot 2^n|\mathtt{p}|$

(2) If p accepts d and $\mathit{time}_{\mathtt{p}}(\mathtt{d}) \leq n$ then
$[\![\mathtt{dtnu}]\!](\mathtt{p.d.nil}^n) = ([\![\mathtt{p}]\!]\mathtt{d}\ \ .\ \mathtt{nil})$
Otherwise,
$[\![\mathtt{dtnu}]\!](\mathtt{p.d.nil}^n) = \mathtt{nil}$

We refer to this interpreter as *inefficient* because its complexity may be exponentially larger than the complexity of the interpreted program. But for this price, it gets rid of nondeterminism. We now describe an implementation of dtnu.

A command choose C1 | C2 is implemented by first running C1, and if it fails or runs too long, backtracking to run C2 instead. Each time that we make the move to C1, we keep all the data needed to continue the computation via C2 in a stack, which we pop when backtracking.

More specifically, the timed interpreter tu shown in Section 5.1 is extended by adding the stack Cont of deferred continuations. Each item on the stack has the form (Cd.Stk.X.Cntr), giving the values of the respective variables for the continuation. To implement this, the STEP macro handles choose C1 | C2 by decrementing Cntr (as a choose costs a time unit), then pushing ((C2.Cd).Stk.X.Cntr) onto the continuation stack, and finally C1 onto the code stack. The main program of the timed interpreter takes care of backtracking to Cont when necessary:

```
read X;                (* X=p.d.nil^n *)
Cd := (hd X).nil;
Cntr := tl tl X;
X := hd tl X;          (* Init. value for x *)
Stk := nil;            (* Computation stack *)
Cont := nil;           (* Continuation stack *)
while Cd and Cntr do {
  while Cd and Cntr do STEP;
  if Cd or not X then {      (* premature stop or rejection *)
    if Cont then       (* backtrack *)
      Cd := hd hd Cont;
      Stk := hd tl hd Cont;
      X := hd tl tl hd Cont;
      Cntr := tl tl tl hd Cont;
      Cont := tl Cont
    else               (* nowhere to backtrack to *)
      X := nil;
  } else               (* ordinary, successful stop *)
    X := X.nil;
}
write X
```

Computation Time. The preparatory part, before the main loop, takes constant time. The main loop mimics the timed interpreter (except for the backtracking). Every computation path followed is constrained by the time limit. Clearly we obtain a worst-case bound on the time by assuming that as many computation paths as possible were tried. The paths describe a binary tree. Every iteration of the internal loop performs STEP on behalf of one of the paths; we can say that each STEP is associated with a node of the computation tree. We now estimate the number of nodes. Every node that corresponds to a marked rule (for brevity: marked node) decreases `Cntr`, and hence on every path there are at most n such nodes. Note that the branching nodes of the tree (corresponding to `choose` instructions) are marked. This shows, that if we remove unmarked nodes from the tree (welding the arcs that lead in and out of them), its topological structure is not changed and its height is bounded by n (the time limit), hence its size by 2^n. We have thus bounded the number of marked nodes.

As observed while proving Theorems 4.5 and 5.2, every unmarked node corresponds to an operation that pushes the code stack while in the marked nodes it is popped. Therefore, every unmarked node is *matched* by a later marked node, except for at most $|\mathtt{p}|$ nodes *on every computation path.* A bound on the number of those "unmatched" nodes is $2^n \cdot |\mathtt{p}|$. As for the matched nodes, their number can be bounded in terms of the matching (marked) nodes as in Theorem 4.5 and we get a bound of $2 \cdot 2^n$.

We conclude that the number of nodes in the tree is $O(2^n \cdot |\mathtt{p}|)$. Every node represents a constant amount of computation time and thus we can find k such that $time_{\mathtt{dtnu}}(\mathtt{p}\,.\,\mathtt{d}\,.\,\mathtt{nil}^n) \leq k \cdot 2^n|\mathtt{p}|$ for all p,d and n.

6 Constant time factors give a proper hierarchy

In this section we use efficient universal programs to prove a series of hierarchy theorems, stating that a constant-factor increase in time allows more problems to be solved. While the ideas of the proofs are not new, the use of the structured language and the efficient universal programs make it easier to carry out the proofs and yield results which are tighter than possible for Turing machines (witness the "constant speedup theorem," [2, 19]).

6.1 Definition of complexity classes

Definition 6.1 *Suppose we are given a programming language* L *with corresponding sets* L-programs *and* L-data *(in particular we consider the language* I *with* I-data $= \mathbb{D}$*). We define (using notation from Section 3.3)* *the class of problems solvable in time bounded by f:*

1. $\mathrm{TIME}^{\mathtt{L}}(f) = \{Acc^{\mathtt{L}}(\mathtt{p}) \mid$
   Deterministic L*-program* p *runs within time* $f\}$

2. $\mathrm{NTIME}^{\mathtt{L}}(f) = \{Acc^{\mathtt{L}}_{\exists}(\mathtt{p}) \mid$
   Nondeterministic L*-program* p *runs within time* $f\}$

6.2 Hierarchy theorems for deterministic time

Definition 6.2 *A function* $f : \mathbb{N} \to \mathbb{N}$*, such that* $f(n) \geq n$*, is called* time constructible in I *if there is an* I*-program that, on input* $\mathtt{nil}^n$*, outputs* $\mathtt{nil}^{f(n)}$*, and its running time is* $O(f(n))$*.*

For brevity the qualification "in I" is omitted in the sequel. Time-constructible functions include many common functions such as n, n^2, 2^n, $n\lceil\log n\rceil$, etc.

Theorem 6.3 *For every time-constructible function* $T(n) \geq n$*, there is a constant* b *such that there are sets in* $\mathrm{TIME}^{\mathtt{I}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}}(T(n))$*.*

Proof: First define program diag informally:

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

The program is completed by filling in the code to compute the size of the input, $|\mathtt{X}|$, and then $\mathtt{nil}^{T(|\mathtt{X}|)}$. For input d, the first computation can be done in time $O(|\mathtt{d}|)$ (Section 3.5), and the second in time $O(T(|\mathtt{d}|))$, since T is time-constructible. Since $T(n) \geq n$, we can assume that the time for both computations is bounded by $cT(|\mathtt{d}|)$ where c is a constant. From Definition 5.1, there exist k, l such that the timed universal program tu of Theorem 5.2 runs in time $time_{\mathtt{tu}}(\mathtt{p}\,.\,\mathtt{d}.\,\mathtt{nil}^n) \leq k \cdot \min(n, time_{\mathtt{p}}(\mathtt{d})) + l \cdot |\mathtt{p}|$.

On input $\mathtt{d} = \mathtt{p}$, program diag runs in time at most $e + l \cdot |\mathtt{p}| + (c+k) \cdot T(|\mathtt{p}|)$ where e accounts for the time beyond computing Timebound and running tu. Defining $b = c + k + l + e$, we have shown $Acc(\mathtt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(b \cdot T(n))$.

Now suppose for the sake of contradiction that $Acc(\mathtt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(T(n))$. Then there exists a program p with $Acc(\mathtt{p}) = Acc(\mathtt{diag})$, and $time_{\mathtt{p}}(\mathtt{d}) \leq T(|\mathtt{d}|)$ for all $\mathtt{d} \in \mathbb{D}$. Consider the application of p to its own code (the usual diagonal argument). The time bound on p implies that $[\![\texttt{tu}]\!](\texttt{p}\,.\,\texttt{p}\,.\,\texttt{nil}^{T(|\texttt{p}|)}) = ([\![\texttt{p}]\!]\texttt{p}\,.\,\texttt{nil})$. If $[\![\texttt{p}]\!]\texttt{p}$ is nil, then $[\![\texttt{diag}]\!]\,\texttt{p} = (\texttt{nil}\,.\,\texttt{nil})$. 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 $Acc(\texttt{diag}) \in \mathrm{TIME}^{\mathtt{I}}(T(n))$ must be false. □

Linear functions $T(n) = an$ are obviously time-constructible, moreover, it is not hard to see that they can all be computed within $cT(n) = can$ time for some universal constant c. Hence we can easily obtain

Corollary 6.4 *There is a constant b such that for all integer $a \geq 1$, there are sets in* $\mathrm{TIME}^{\mathtt{I}}(abn) \setminus \mathrm{TIME}^{\mathtt{I}}(an)$.

Thus the class of linear-time decidable sets contains an infinite hierarchy.

Another simple extension to the theorem's proof can be used to provide an infinity of inputs on which the alleged program p errs. These inputs would be obtained by padding p with idle commands to an arbitrarily large size. This shows, that there are sets in $\mathrm{TIME}^{\mathtt{I}}(b \cdot T(n))$ which cannot be recognized within time $T(n)$ even if a finite number of exceptions is excused. Yet another extension shows that the set in $\mathrm{TIME}^{\mathtt{I}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}}(T(n))$ can be guaranteed to be a *tally set*: a set S of trees such that the inclusion of a tree in S depends only on its size. This is achieved in essence by mapping program codes into integers so that instead of inputting the tree representing p we input a tree whose size represents p. The interested reader may find this technique applied (with more detail) in some papers about hierarchy theorems [34, 35].

Finally note, that given a program for constructing $T(n)$, of known complexity, the constant b of the last theorem can be effectively computed. In fact, by careful coding and tedious calculations it was shown [6] that corollary 6.4 holds with $b = 232$ for all a, and with $b = 127$ for $a \geq 106$.

The next corollary shows that "the constant that matters" can be as near to one as desired. In other words: there are programs that cannot be sped up even by a very small given constant. However in contrast with the former theorem, this result is not constructive.

Corollary 6.5 *For every constant $\varepsilon > 0$, and time-constructible $T(n)$, there is a constant $c \geq 1$ such that* $\mathrm{TIME}^{\mathtt{I}}((1+\varepsilon)c \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}}(c \cdot T(n))$ *is not empty.*

Proof: Fix the function $T(n)$ and the constant ε. For $k = 1, 2, \ldots$ define

$$S_k = \mathrm{TIME}^{\mathtt{I}}((1+\varepsilon)^k \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}}((1+\varepsilon)^{k-1} \cdot T(n))$$

and let b be given by the last theorem; $\mathrm{TIME}^{\mathrm{I}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathrm{I}}(T(n))$ is not empty. Let $\ell = \lceil \log_{1+\varepsilon} b \rceil$. Then

$$\mathrm{TIME}^{\mathrm{I}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathrm{I}}(T(n)) \subseteq \bigcup_{k=1}^{\ell} S_k$$

hence the last union is not empty. Therefore, there is a k such that S_k is non-empty. Let $c = (1+\varepsilon)^{k-1}$; the corollary is satisfied. □

6.3 Hierarchy theorems for nondeterministic time

Seiferas, Fischer and Meyer [34] proved a time hierarchy theorem for nondeterministic Turing machines. A tighter hierarchy was later proven by Žák [39]. Using the proof strategy of [39] we are able to prove a tight hierarchy theorem for nondeterministic `I` programs.

Theorem 6.6 *For every time-constructible non-decreasing function $T(n) \geq n$, there is a constant b such that there are sets in $\mathrm{NTIME}^{\mathrm{I}}(b \cdot T(n+2)) \setminus \mathrm{NTIME}^{\mathrm{I}}(T(n))$.*

Note that the function $b \cdot T(n+2)$ may be more than a constant multiple of $T(n)$; this depends on the growth rate of T. However, for functions T that grow at most as fast as exponentials, we have $T(n+2) = O(T(n))$ so the hierarchy is constant-factor tight. The size of a tree is always an odd number, hence the reference to $T(n+2)$ (and not $T(n+1)$ as in [39]).

Proof: Again we use a diagonalization argument, but this one is a little more complicated. We use the following program. As before, we handle integer quantities in our program, which are represented as lists of `nil`'s in the standard way. As a form of "syntactic sugar" or "macro programming" we apply arithmetic operations to these numbers: e.g., $\leq$ for comparison, which can clearly be implemented with a loop in linear time, and log for computing the base-2 logarithm, which can be performed in linear time too although this is perhaps less obvious (the reader may like to figure out the details).

Note that the program uses both universal programs `tnu` and `dtnu`. To understand the program, think of its input as $\texttt{p.nil}^n$ for a program `p`.

```
read X;
Timebound := T(|X| + 2);
Smallbound := T(|hd X| + 2);
Log := ⌊log Timebound⌋ − ⌊log |hd X|⌋;
```

```
if Smallbound ≤ Log then
  X := dtnu(hd X,(hd X).nil,Log);
  if hd X then X := nil else X := (nil.nil)
else
  X := tnu(hd X,(hd X).(nil.(tl X)),Timebound)
write X
```

Let n be the input size. The size of a tree can be computed in linear time. So by the time-constructibility of T, the second and third lines can be replaced by commands that run in time $O(T(n+2))$, say at most $cT(n+2)$ where c is a constant. Let a be a constant such that the computation of Log can be performed within time $aT(n+2)$. Let k be such a constant that both the efficient universal program tnu and the inefficient one dtnu run for at most k times the value of Timebound (we leave it to the reader to verify that, as a result of the time constraints on the two interpreters, such a k can indeed be found).

On input x, program diag runs in time at most $(e+a+c+k)\cdot T(n+2)$ where e accounts for the time spent outside the program parts mentioned above. Defining $b = e+a+c+k$, we have shown $Acc_\exists(\texttt{diag}) \in \mathrm{NTIME}^{\mathtt{I}}(b \cdot T(n+2))$.

Let us assume that $Acc_\exists(\texttt{diag}) \in \mathrm{NTIME}^{\mathtt{I}}(T(n))$. Then there exists a program p with $Acc_\exists(\mathrm{p}) = Acc_\exists(\texttt{diag})$, and $time_{\mathrm{p}}(\mathrm{d}) \leq T(|\mathrm{d}|)$ for all $\mathrm{d} \in I\!D$. Consider the application of p to inputs of the form $(\mathtt{p.nil}^j)$, $j \geq 0$ (recall that $\mathtt{nil}^0$ is nil, and $\mathtt{nil}^{j+1} = \mathtt{nil.nil}^j$, so $|\mathtt{nil}^j| = 2j+1$). For such inputs we have:

$$\begin{aligned} \texttt{Timebound} &= T(|\mathrm{x}|+2) = T(|\mathrm{p}|+2j+4) \\ \texttt{Smallbound} &= T(|\mathrm{p}|+2) \end{aligned}$$

Let j_0 be the smallest value of j such that $T(|\mathrm{p}|+2j+4) \geq 2^{T(|\mathrm{p}|+2)} \cdot |\mathrm{p}|$. Assume $j_0 > 0$ (the case $j_0 = 0$ is easier and omitted). Consider first inputs of the form $(\mathtt{p.nil}^j)$, $j < j_0$. On such input, the condition Smallbound $\leq$ Log is not satisfied, and the program runs $\mathtt{tnu(p,p.nil}^{j+1}\mathtt{,Timebound)}$. By our assumption,

$$time_{\mathrm{p}}(\mathtt{p.nil}^{j+1}) \leq T(|\mathrm{p}|+2j+4) = \texttt{Timebound},$$

hence tnu will simulate p correctly on this input. It follows that

$$\mathtt{p.nil}^j \in Acc_\exists(\texttt{diag}) \iff \mathtt{p.nil}^{j+1} \in Acc_\exists(\mathrm{p}).$$

However we assumed that $Acc_\exists(\mathtt{p}) = Acc_\exists(\mathtt{diag})$, so

$$\mathtt{p.nil}^j \in Acc_\exists(\mathtt{p}) \iff \mathtt{p.nil}^{j+1} \in Acc_\exists(\mathtt{p})\,.$$

Since this is true for $j = 0, 1, 2, \ldots, j_0 - 1$ we clearly have

$$\mathtt{p.nil}^0 \in Acc_\exists(\mathtt{p}) \iff \mathtt{p.nil}^{j_0} \in Acc_\exists(\mathtt{p})\,. \tag{3}$$

We now consider the input $(\mathtt{p.nil}^{j_0})$. The condition $\mathtt{Smallbound} \le \mathtt{Log}$ is satisfied, and the program runs dtnu(p,p.nil,Timebound). Now, the running time of p on input p.nil (if accepted) is assumed to be at most

$$T(|\mathtt{p.nil}|) = T(|\mathtt{p}| + 2) = \mathtt{Smallbound} \le \mathtt{Log}\,.$$

According to the definition of dtnu, it will simulate p correctly on input (p.nil). Program diag proceeds by inverting its result. We thus obtain:

$$(\mathtt{p.nil}^{j_0}) \in Acc_\exists(\mathtt{diag}) \iff (\mathtt{p.nil}) \notin Acc_\exists(\mathtt{p})\,.$$

Since we assumed that $Acc_\exists(\mathtt{p}) = Acc_\exists(\mathtt{diag})$ we obtain

$$(\mathtt{p.nil}^{j_0}) \in Acc_\exists(\mathtt{p}) \iff (\mathtt{p.nil}) \notin Acc_\exists(\mathtt{p})$$

contradicting (3). $\square$

6.4 A note on time-constructible functions

We conclude this section with a theorem that states that if $T(n)$ is a function that describes the running time of some program on inputs of the form $\mathtt{nil}^n$, then $T(n)$ is time constructible.

Theorem 6.7 *If there is an* I *program* p *such that on input* $\mathtt{nil}^n$*, its running time is* $f(n)$*, then* $f(n)$ *is time-constructible in* I.

PROOF: We obtain a program p′ to compute $f(n)$ by modifying the program p. A new variable Time is used to count the running time of p. It is initialized to nil. When p halts, its value will be $\mathtt{nil}^{f(n)}$.

To achieve that, every command of p is followed by a command to add the appropriate time figure to Time. This is not hard to do as the time for evaluating any given expression in our language is a constant, actually the size of the expression. Since the time consumed by the added command is proportional to the time of the original command, the running time of p′ will be $O(f(n))$, as required. $\square$

The reverse statement: if $f(n)$ is time-constructible then it is the running time of some program (on inputs as above), is not true in general. In fact, Theorem 6.3 (when extended to tally sets) may be used to give a counter-example to this statement (we leave details to the reader). We point this out because for Turing machines the reverse statement does hold, following from the constant speedup theorem [2].

7 Variations on the language `I`

In this section we investigate the effects of extending our language with stronger control on storage, that is, giving it access to the *graph structure* that in the `I` language is invisible to the programmer (see Section 3.2). The reason that we are interested in such extensions are that `cons`, `hd` and `tl` alone are too weak for efficient programming. The extensions that we consider are in fact included in practice in LISP in order to gain efficiency. We discuss two extensions: pointer comparison and selective update (also called destructive update, or mutation).

7.1 The languages $\mathtt{I}^{eq}$ and $\mathtt{I}^{su}$

The data structuring facilities of our language `I` are modelled after popular list-processing languages such as LISP and its descendant Scheme. The standard implementation of such facilities uses a pair of pointers to represent a cons cell; these point to the memory locations of its head and tail. Because of data sharing, the storage structure that results is a DAG.

Thinking of the head and tail constituents of a cell as pointers to other cells (or to atoms, also stored in memory) it is natural to extend the language by an operation to test the identity of pointers.

Formally, the language $\mathtt{I}^{eq}$ is obtained from `I` by adding a predicate `eq`. The expression `(eq E1 E2)` evaluates to a fixed non-nil value (say `nil.nil`), indicating truth, if both expressions `E1` and `E2` are equal atoms (in the basic language, this means they are both `nil`) or if both evaluate to the same node in the DAG structure. Thus the following two programs yield different results:

```
X := (nil.nil);
X := cons X X;
Y := (eq (hd X)(tl X))        (* yields true *)
```

and

```
X := ((nil.nil).(nil.nil));
Y := (eq (hd X) (tl X))        (* yields false *)
```

Note that for an I program there is no difference between the values that X obtained in the two code segments.

Selective update is present in LISP and Scheme under the names `rplaca`, `rplacd` and `setcar`, `setcdr` respectively. These operations allow the contents of a cons-cell to be modified. We incorporate this extension into our language by generalising the assignment statement to the form:

$$\texttt{LE := E}$$

Where LE (a left-hand expression) is defined by

$$\begin{array}{rcl} \mathrm{LE} & ::= & \mathrm{X} \\ & | & \texttt{hd}\ \mathrm{LE} \\ & | & \texttt{tl}\ \mathrm{LE} \end{array}$$

An example of the extended assignment is:

$$\texttt{hd tl X := X}$$

This alters the head component of `tl X`, provided it is a cons cell; otherwise the command is "invalid" and does nothing. Clearly, if the command is valid, it will create a cyclic structure. Thus in this language, the storage DAG turns into a (possibly cyclic) directed graph.

We observe that this extension is at least as strong as the previous:

Theorem 7.1 *The predicate* `eq` *can be simulated by a constant number of* $\mathtt{I}^{su}$ *operations.*

Proof: It will be convenient here to make use of several variable names, using the standard technique; this is not essential to the result. We thus assume that our goal is to evaluate `(eq X Y)` where X and Y are variables. We give a high-level description of the procedure to carry this out:

1. If either of X or Y equals `nil`, the test succeeds if and only if the other equals `nil` as well. (This may be generalised for a multi-atom language).
2. Similarly for `(nil.nil)`.
3. We now assume without loss of generality that X as well as `hd X` are different from `nil` (the case of `tl X` being symmetric). If `hd Y` does

equal `nil`, the test fails. Otherwise (both of `hd X` and `hd Y` are non-nil), let `Z` be a temporary variable. Set `Z` to `hd X`, then set `hd X` to `nil`. The answer to `(eq X Y)` is `true` if this causes `hd Y` to also become `nil`. After checking this, `hd X` is restored from `Z`. □

Thanks to this theorem, we may assume for our convenience that the $\mathtt{I}^{su}$ language includes the `eq` predicate as well. So we have three languages of non-decreasing power: `I`, $\mathtt{I}^{eq}$ and $\mathtt{I}^{su}$ (curiously, most definitions of LISP include selective update but state that the equality predicate must only be used to compare atoms).

The reader may justly ask what justifies the introduction of three different models. To warrant this multiplicity, each variant has to have its own advantages, or there may be something to be gained from studying the relationship among the variants.

Arguments in favor of $\mathtt{I}^{su}$. The operations of selective update and equality testing correspond to assignment and comparison operations in imperative languages with pointers. When programming in such languages it seems almost obvious that these operations are essential for efficiency and should be included. In fact, some constructions used earlier in this paper can be strengthened with the help of these operations; this is shown later in this section. Studying the $\mathtt{I}^{su}$ model continues an existing interest in the power of programs based on pointer manipulation. In fact, it almost coincides with an abstract machine model that has received considerable attention, Schönhage's *storage modification machine* [32, 4, 13, 24, 36, 37].

Arguments in favor of `I` and $\mathtt{I}^{eq}$. As we already remarked, the semantics of the three languages is increasingly complex (in the order of their presentation). Simple semantics are an advantage from the point of view of language implementation, as well as for reasoning about programs. Regarding implementation, a notable example is the problem of garbage collection that arises in the presence of cyclic structures. We next point out some known differences related to reasoning about programs.

First, only the "pure" language `I` has the property that if two expressions yield the same observed value then they can be used interchangeably. Here by "observed value" we mean a tree, such as `((nil.nil).(nil.nil))`. This value has been used in an example earlier in this section to illustrate that the above property does not hold for $\mathtt{I}^{eq}$.

$\mathtt{I}^{su}$ programs are even more difficult to understand or analyze, because when two expressions evaluate to the same node in the storage graph, an assignment that uses one will also affect the other. A salient result of this *aliasing* phenomenon is that after executing the command:

```
hd X := hd hd Y
```

it is possible that `hd X` will not be the same as `hd hd Y` (this has been called "the paradox of assignment" [36], and can also be demonstrated with one variable but longer expressions).

Effect of the changes on complexity. Our three languages being different in their semantic properties, or programming flexibility, doesn't necessarily mean that as models for *complexity* there is any difference. A challenging question for further research is: do the language extensions allow any problems to be solved faster? At the moment, a proof of this fact has only been established in a somewhat restricted framework, regarding on-line programs with an infinity of atoms, see Pippenger [25]. Fuller answers to this question would be of great interest to the study of programming languages, since in a sense it concerns the *price for programming in a purely functional style*. We remark that the known upper bound on the gap between time complexity in I and $\mathtt{I}^{su}$ is logarithmic: any $\mathtt{I}^{su}$ program of running time $t(n)$ can be replaced by an I program of time $O(t(n) \log t(n))$ [25].

7.2 Robustness of previous results

It is quite straightforward to see that the results that we have proved for language I also hold for the extended languages. In particular:

- Each of these languages may be endowed with more atoms and more variables. Such extended programs may be translated back into the one-atom, one-variable language with a slow-down that only depends on the number of added atoms or variables.

- The various universal programs with efficiency guarantees, given in Section 4, exist for each of the three languages (the interpreter for L-programs is always written in L). The interpreters for $\mathtt{I}^{eq}$ are obtained by extending the STEP macro with a rule that implements eq by means of eq. In the case of $\mathtt{I}^{su}$, rules have to be added for handling assignment with a general left-hand side expression. We omit the details which are not difficult.

- The time hierarchy theorems transfer to the extended languages with no change in their proofs.

In fact, for the extended languages we were able to extend the hierarchy theorems as shown in the next two subsetions.

7.3 Hierarchy theorem: unboundedly many atoms

In this section, we extend the hierarchy theorem for deterministic time (Theorem 6.3) to a language with an unbounded number of atoms. Let language $\mathtt{I}^{eq}$+ be identical to $\mathtt{I}^{eq}$, except that data and programs may contain any atom from an infinite set $a_1, a_2, \ldots$ The language semantics is the same except for adding the functionality of comparing atoms by `atom=?`.

The difficulty in handling this extension lies in the construction of an efficient universal program. If we only extend the language with a fixed number of "new" atoms, it is easy to write an efficient universal program as for the case of a single atom. But the straightforward approach fails if the number of extra atoms is unbounded. We will solve this problem using the power of `eq`. In order to handle programs that contain data values with unboundedly many atoms in a uniform way, we encode $\mathbb{D}_{\{a_1,a_2,\ldots\}}$ in $\mathbb{D}$ using the scheme described in Section 3.6.1. The atom a_i is encoded as $\mathtt{nil}^i$. Note that cons pairs `(d1.d2)` are also encoded in a special way, $\mathtt{((nil.nil).}\overline{\mathtt{d1}}\mathtt{.}\overline{\mathtt{d2}}\mathtt{)}$, so that the head of any subtree signifies whether it encodes a pair or an atom.

Special atoms used in the syntax of the language (e.g., `while`) will be included among the a_i.

We now present a "multi-atom timed universal program," an $\mathtt{I}^{eq}$ program to interpret encoded $\mathtt{I}^{eq}$+ programs. The interpreter assumes that the program will only be applied to inputs in $\mathbb{D}_{\mathtt{nil}}$. Thus only the program code may use different atoms. Yet the input to the program will have to be encoded using the same scheme we use for multi-atom trees, so that a given tree will be represented in the same way be it input or a program constant.

Lemma 7.2 *There is a program* `mau`*, written in* $\mathtt{I}^{eq}$*, with input* $\mathtt{(p.d.nil}^n\mathtt{)}$ *where* $\mathtt{p}, \mathtt{d} \in \mathbb{D}$*,* `p` *is an encoding of an* $\mathtt{I}^{eq}$+ *program, and* `d` *an encoding of an arbitrary tree in* $\mathbb{D}$*, such that:*

If $\mathit{time}_{\mathtt{p}}(\mathtt{d}) \le n$ *then* $[\![\mathtt{mau}]\!](\mathtt{p\ .\ d\ .\ nil}^n) = ([\![\mathtt{p}]\!]\mathtt{d\ .\ nil})$

If $\mathit{time}_{\mathtt{p}}(\mathtt{d}) > n$ *then* $[\![\mathtt{mau}]\!](\mathtt{p\ .\ d\ .\ nil}^n) = \mathtt{nil}$.

Program `mau` *runs in time bounded by* $k \cdot \min(n, \mathit{time}_{\mathtt{p}}(\mathtt{d})) + l \cdot |\mathtt{p}|$*, for some constants* k *and* l*.*

Proof: Program mau starts by preprocessing the program code. This process decodes the encoding of pairs and atoms, and whenever an encoding of atom a_i is encountered, it is replaced by a reference to a unique cons cell c_i (we give more details below). After this processing, we obtain program code that can be interpreted (and timed) as an ordinary (single atom) $\mathtt{I}^{eq}$ program, except that we add a new rule for the additional operation `atom=?`. Because every atom is represented by a unique cell, this operation can be implemented by means of `eq` and in time independent of the number of atoms in the program.

The interpreter runs in time bounded by $k \cdot \min(n, \mathit{time}_{\mathtt{p}}(\mathtt{d})) + l' \cdot |\mathtt{p}|$ for constants k and l'. Our goal is to perform the preprocessing in time $O(|\mathtt{p}|)$, so that the total running time of mau be as stated in the theorem.

The special cons cells c_i will be taken from a list of $|\mathtt{p}|$ `nil`'s, held in a variable `Lst`. The cell c_i is the ith cell in the spine of the list:

$$\begin{array}{ccccccc} & c_1 & & c_2 & & c_3 & \\ \mathtt{Lst}\longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow\cdots \\ & \downarrow & & \downarrow & & \downarrow & \\ & \mathtt{nil} & & \mathtt{nil} & & \mathtt{nil} & \end{array}$$

Creating this list is the same as computing the size of p, which can be done in linear time (Section 3.6.1). Note that every atom a_i is encoded in p as $\mathtt{nil}^i$; hence $|\mathtt{p}|$ is at least as large as the largest i such that a_i appears in p. This shows that the list is long enough to be used for the c_i's.

The preprocessing is done by the program shown in Figure 4 at the end of the paper. This program implements a tree transformer that decodes an encoded tree over $\mathbb{D}_{\{a_1,a_2,\ldots\}}$ and replaces the references to a_i as required. A rough outline of the program is as follows: decompose the input tree down to its leaves, then pair the results.

Verifying that the total time for the preprocessing stage is linear in $|\mathtt{p}|$ is achieved by charging every iteration of every loop in Figure 4 to a distinct node in p. Since this is routine we omit the details. □

Theorem 7.3 *Let $T(n) \geq n$ be time-constructible. There is a constant b such that there are sets in* $\mathrm{TIME}^{\mathtt{I}^{eq}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}^{eq}+}(T(n))$.

Note that the theorem states that the set in question can be decided in time $bT(n)$ without the use of extra atoms, while even with their use it cannot be decided in time $T(n)$.

The proof is identical to that of Theorem 6.3, except that the program input is now an encoded $\mathtt{I}^{eq}$+ program and is executed using mau.

We remark that the same extension can also be applied to the other theorems in Section 6. We also remark that the proof (or, much less likely, disproof) of such a theorem without the use of eq rests open.

7.4 Hierarchy theorem: unboundedly many variables

Adding extra variables to I was also discussed in Section 3.6.1. If the number of added variables is fixed for all programs, the extended language can be translated into the base language with a constant-factor slowdown; but if the number of extra variables is unbounded, no such translation is known. As in the last subsection we solve this problem by creating a special interpreter for encoded multiatom programs. Here we make use of selective update.

Define $\mathtt{I}^{su}$++ to be $\mathtt{I}^{su}$ plus unboundedly many atoms *and* unboundedly many variables. The concrete syntax is extended by the construct (var $\mathtt{nil}^i$) which denotes "variable number i". We encode $\mathtt{I}^{su}$++ programs as trees in $\mathbb{D}$ using the standard technique, as in the last subsection.

Lemma 7.4 *There is a program* mvau*, written in* $\mathtt{I}^{su}$*, such that, on input* $(\mathtt{p.d.nil}^n)$ *where* $\mathtt{p}, \mathtt{d} \in \mathbb{D}$*,* p *is an encoding of an* $\mathtt{I}^{su}$++ *program, and* d *an encoding of an arbitrary tree in* $\mathbb{D}$*, it runs in time bounded by* $k' \cdot (|\mathtt{p}|^2 + \min(n, \mathit{time}_{\mathtt{p}}(\mathtt{d})))$*, for some constant* k'*, and satisfies:*
If $\mathit{time}_{\mathtt{p}}(\mathtt{d}) \leq n$ *then* $[\![\mathtt{mvau}]\!](\mathtt{p\ .\ d\ .\ nil}^n) = ([\![\mathtt{p}]\!]\mathtt{d\ .\ nil})$

If $\mathit{time}_{\mathtt{p}}(\mathtt{d}) > n$ *then* $[\![\mathtt{mvau}]\!](\mathtt{p\ .\ d\ .\ nil}^n) = \mathtt{nil}$

Proof: Program mvau operates very similar to program mau above, so we only note the differences. As above, we replace every occurrence of the atom a_i in p by a reference to a cons cell c_i, so operations on atoms can be carried out efficiently. In addition we replace variable numbers (the second part of (var $\mathtt{nil}^i$)) with a reference to v_i, which like c_i is found in the ith position of a list of length $|\mathtt{p}|$. The hd field of each v_i holds the value of the corresponding variable.

$$
\begin{array}{ccccccc}
 & v_1 & & v_2 & & v_3 & \\
\longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow \mathtt{nil} \\
 & \downarrow & & \downarrow & & \downarrow & \\
 & \text{value} & & \text{value} & & \text{value} &
\end{array}
$$

Creating this linkage requires just a minor change to the preprocessing program, which we omit. After preprocessing, an efficient timed self-interpreter for $\mathtt{I}^{su}$ is run, extended to interpret atom=? by means of eq (as in the

last subsection), and also to handle variable references. When the construct (`var` v) is encountered, v is known to be one of the v_i, and the interpreter uses `hd` v to access or modify the variable's value. □
As usual, this implies

Theorem 7.5 *Let $T(n) \geq n$ be time-constructible. There is a constant b such that there are sets in* $\mathrm{TIME}^{\mathtt{I}^{su}}(b \cdot T(n)) \setminus \mathrm{TIME}^{\mathtt{I}^{su}++}(T(n))$.

As for the previous result, we can apply the same extension the other theorems in Section 6 as well, and we pose as an open problem their proof in the basic `I` language.

8 A problem complete for Nondeterministic Linear Time

The class of problems solvable in linear time is known to be difficult to study, mainly because the notion of linear time is very model dependent [14, 26]. However, we maintain that the languages presented in this paper are powerful enough (yet not too much) so that linear-time computation in these languages is of interest. The classes $\mathrm{LIN}^{\mathtt{L}}$, of sets decidable by `L`-programs in deterministic linear time, and similarly $\mathrm{NLIN}^{\mathtt{L}}$, for non-deterministic time, exhibit the typical properties of well-studied complexity classes, in particular we do not know whether $\mathrm{LIN}^{\mathtt{L}} = \mathrm{NLIN}^{\mathtt{L}}$ for any language here considered, and in this section we provide a natural example of a decision problem complete for $\mathrm{NLIN}^{\mathtt{L}}$ under deterministic linear-time reductions. For notational convenience we will only treat `I` and omit the language name in the notations LIN, NLIN, though the proposition and its proof can be carried out using $\mathtt{I}^{eq}$ or $\mathtt{I}^{su}$ with no essential change.

We do not elaborate on the notions of "linear time reduction" and "complete problem" which would be obvious to a reader familiar with the topic of NP completeness. The details can be found in [19]. The major consequence of a problem being NLIN-complete is that if this problem can be solved in deterministic linear time then NLIN=LIN (in fact if it could be solved in polynomial time then we would have NLIN⊆PTIME, also an open problem). For further discussion on complete problems for "small" non-deterministic time classes, see also [12, 14, 31].

Let `NI`-prog denote the set of non-deterministic `I` programs.

Theorem 8.1 *Problem* WFA *is complete for* NLIN, *where*

$$\mathrm{WFA} = \{ (\mathtt{p}.\mathtt{d}) \mid \mathtt{p} \in \mathtt{NI}\text{-prog } \textit{is while-free and } \mathtt{d} \in Acc_{\exists}(\mathtt{p}) \}.$$

PROOF: We first show that WFA $\in$ NLIN. To this end, modify the nondeterministic universal program nu as follows. First, check that input p encodes a while-free program, and then continue with executing p on input d as before. The checking can be done in time linear in $|\mathtt{p}|$, and while-freeness implies that $\mathit{time}_{\mathtt{p}}(\mathtt{d}) \leq |\mathtt{p}|$ regardless of d. Due to the efficiency of nu, we can recognize WFA in time linear in $|(\mathtt{p}\,.\,\mathtt{d})|$.

To show completeness, consider a program p such that $P = \mathit{Acc}_{\exists}(\mathtt{p}) \in$ NLIN. Let a, b be integers such that $\mathit{time}_{\mathtt{p}}(\mathtt{d}) \leq a \cdot |d| + b$. Given d, define $f_p(\mathtt{d}) = (\mathtt{q}\,.\,\mathtt{d})$ where q is the following program. Here $\mathtt{STEP}^n$ stands for n copies of the code for the (nondeterministic) STEP macro:

```
read X;
Cd := p.nil;       (* Code to be executed *)
Stk := nil;        (* Computation stack *)
STEP^{a·|d|+b};
write X
```

Clearly, $f(\mathtt{d})$ can be computed from d in $O(a \cdot |\mathtt{d}| + b)$ time, i.e., in (deterministic) linear time. Program q is while-free, and it simulates p on d for $a \cdot |\mathtt{d}| + b$ steps. This is long enough for p to accept (if $\mathtt{d} \in P$), so

$$\mathtt{d} \in \mathit{Acc}_{\exists}(\mathtt{p}) \iff \mathtt{d} \in \mathit{Acc}_{\exists}(\mathtt{q}) \iff (\mathtt{q}.\mathtt{d}) \in \mathrm{WFA}$$

thus P is reduced to WFA. $\Box$

9 Some remarks on efficient metaprogramming

Metaprogramming is the manipulation of programs by programs. Since the advent of high-level languages, this type of programming has been pervasive. Certain programming languages are specifically tailored to efficient metaprogramming: LISP and its descendants are an example. Among the different kinds of metaprograms, the interpreter is perhaps the conceptually simplest. Interpreters played a role in theory since the introduction of the universal Turing machine. More involved metaprogramming, where programs are output by the metaprogram, has also appeared early in computability theory, for example in many proofs of undecidability using reduction from the halting problem. As long as computability only is considered, the efficiency of metaprogramming could be neglected. But efficiency does matter when metaprogramming is used in complexity theory (and, of course, in practice). The languages that we presented in this paper are tailored towards convenient and efficient metaprogramming. In this section we will argue that this is a desirable feature of a model for complexity.

In this paper we have proved some constant-factor tight hierarchy theorems. The techniques of our proofs are standard, the same that have been applied before to Turing machines; with these techniques, the tightness of hierarchy theorems is directly determined by the efficiency of the available universal programs. Thus one benefit of having an efficient interpreter is a tight hierarchy theorem. As discussed in the introduction, we consider such theorems to be not only "aesthetically pleasing" but also a better representation of reality. Furthermore, regardless of its usage in proofs, efficient interpretation is a vital ingredient of realistic computing, making its first appearance in the form of the stored-program computer. This alone should make it natural to look for this property in theoretical models. An abstract machine that models the stored-program computer has been described under the name of RASP (random access stored program computer) [15, 10]. Sudborough and Zalcberg [35] used the possibility of efficient interpretation in this model to prove constant-factor tight time hierarchies. We note however that for results equivalent to those proved in this paper, the ordinary RAM (without a stored program) will do, because it can still store the interpreted program in random access memory; it is not necessary for the proofs that the interpreter itself residess in memory.

Regarding the random access machine, authors often claim that because of its unbounded word length, a logarithmic cost measure is appropriate for making its time complexity realistic. This leads to the logarithmic-cost RAM model. Regrettably, the logarithmic-cost RAM does not possess an efficient self-interpreter in our sense, i.e., having a program-independent slowdown. The reason, naturally, is the cost of accessing the program's instructions. Here the RASP model makes a difference, because in this model execution time includes the time to access instructions.

A more involved result where efficient interpretation plays a role is known as Levin's Theorem [23, 13, 30, 19]. This interesting theorem asserts the existence of time-optimal algorithms for a certain class of computational problems. Informally, the definition of the class is "problems where it is easier to verify a solution than to find it," obviously a very wide class. This is a "countertheorem" to Blum's Speedup Theorem: the latter states that there exist problems with no best algorithm, but the problems given are rather contrived. The idea in the proof of Levin's theorem is to solve the problem by rapidly constructing candidate programs and testing them on a given input. Thus metaprogramming and its efficiency are essential. Proofs of the theorem have been described by Gurevich [13] for Kolmogorov-Uspenskii Machines and by Schnorr for random access machines [30]. A less sketchy proof has been given by the authors of this paper in [19], using language I. The efficiency of the I self-interpreter makes the proof work, and the convenience of metaprogramming makes it possible to give the proof in complete detail.

10 Extensions and related work

Changes of programming language. As mentioned in the introduction, we believe that one advantage of adopting the programming-language approach is that it gives a framework for studying questions relating complexity and programming languages, such as the effect of changes in programming style on the efficiency of programs.

Our language I is characterized by its data type (trees) and programming style (imperative, hence its name). Jones [18, 19] defines a simple first-order pure-functional language F whose programs have one recursively defined function of one variable. As in I, more variables and more functions can be added at a multiplicative constant-factor cost, making the language easier to use. It is shown that an efficient interpreter for I can be written in F and vice versa. It follows that any I program can be replaced by an equivalent F program such that the running time of the F program is at most a constant factor times the running time of the I program. Moreover, this constant factor is independent of the particular program. The same holds in the other direction. This is a particularly strong form of equivalence, showing that time complexity in the pure-functional language is essentially the same as in the imperative one, even if constant factors are considered to matter. For example the constant-factor hierarchies hold for F as well.

Higher-order functional programs (where functions can be passed to and returned from functions) are considered by Rose [29], who shows that a high-order language $\mathtt{F}^{ho}$ is also equivalent to I in the above sense. We remark that Rose deals with an abstract machine model (the Categorical Abstract Machine) rather than directly with the language; the considerations for doing so are discussed in her paper.

Another interesting feature is *reflection*: the ability of a program to refer to its own code. Reflection has been used as a tool in computability theory [19, 21, 28] and in complexity theory [8, 33]. Jones [18, 19] defines a reflexive version $\mathtt{I}^{\uparrow}$ of the I language and shows that once again this language is equivalent to I in the above strong sense.

A slightly weaker equivalence is obtained for an unstructured ("flow chart") language, given in [19] under the name of GOTO. This language (whose operations on data are still the same as in I) is equivalent to I in running time up to a *program-dependent* constant factor. This means that constant-factor-tight results do not necessarily hold for both languages, and in particular we have not obtained constant-factor tight hierarchy theorems for GOTO. However, for $\mathtt{GOTO}^{su}$ such theorems can be proved using techniques similar to those of Section 7.

Taking a broader look at the theoretical literature, we find that many separation results between abstract machines can be interpreted as comparisons of language features. There are many such results so we will not attempt to list them here. One example is the comparison of "pointers versus addresses" [5, 3], which is more naturally interpreted as a question about programming languages than about machines. Another is Cook's fast simulation of pushdown automata by random access machines [9, 17]. A collection of other comparisons can be found in van Emde Boas' survey [37].

String input and conventional computational models. The usage of $\mathbb{D}$ as the input/output domain is certainly a non-orthodox feature of our language. Standard practice in complexity theory is to consider the input and output as strings. This difference can easily be reconciled by extending our language with bit input/output (a general alphabet is less convenient, however it is well known that this does not matter in most cases).

The extension is obtained by adding instructions to read a bit and to output one. Then complexity in our language can be directly compared with that in known computational models that use string input/output. To do that, we fix a correspondence between trees and strings. We found the *preorder encoding* of trees to be particularly convenient. Using this encoding, a string that encodes a tree can be transformed into that tree using linear time in I. Similarly, a tree's encoding can be output in linear time. Another possibility is to encode a string as a list of "bits," using special atoms 0 and 1. Once again the translation for input or output takes linear time. As a result, all statements about time complexity, as long as the time is at least linear, do not depend on the choice of the I/O domain.

Once our language is supplied with bit input/output it becomes possible to directly compare its computational power with that of conventional computational models, i.e., abstract machines. In fact, the variant $\mathtt{I}^{su}$ is almost identical to a model known as the Storage Modification Machine [32]. This model is proved in [32] to be real-time equivalent to a particular sort of RAM, the Successor RAM. This means that time complexity on the three models is the same up to a program-dependent constant factor. Both models belong to a cluster of models that can simulate each other in near-linear time $O(t(n)\log^k t(n))$; the cluster also contains some special types of Turing machines and other restricted RAMs, for details see [14].

Tight hierarchy theorems. As mentioned in Section 9, hierarchy theorems have been proved for random access machines that are tighter than the well-known theorems for Turing machines [16, 2]. Close hierarchy results were proved for the Storage Modification Machine [24] and for k-tape Turing machines for $k \geq 2$ [11]. In both models, it is shown that TIME($o(f(n))$) is smaller than TIME($f(n)$. However a constant-factor separation is not obtained because the interpreter has a program-dependent slowdown.

Žák [39] proves a hierarchy theorem for non-deterministic Turing machines (our Theorem 6.6 follows the lines of his work). While for ordinary Turing machines a constant-factor separation is not achieved, he also considers a somewhat artificial model in which the Turing machine has to be processed by a single fixed interpreter, and the cost of interpretation is measured. In this model he shows that a constant-factor separation holds.

Finally, Regan [27] presents a verstile model of computation, or rather a model family, that ranges between weak models that like Turing machines have linear speedup, and strong models where linear speedup does not hold.

Subrecursive languages. An interesting connection between programming languages and complexity is the connection between classes of sets that can be recognized with restricted resources (complexity classes) and classes of sets recognized by restricted programming languages. Jones [20] shows that the classes LOGSPACE and PTIME can be characterized by means of computability in restricted variants of our languages `I` and `F`. These classes are defined with respect to string input; a research question that naturally sprang out is to consider tree input. Some work in this direction is reported by Ben-Amram and Petersen [7]. Voda [38] uses $\mathbb{D}$ as the data type for defining the class of primitive recursive functions and its sub-hierarchies, and compares those with standard definitions based on natural numbers. He also mentions connections to complexity classes.

11 Conclusion

In this paper we described a simple programming language based on the manipulation of binary trees (or LISP S-expressions), and used it as a tool for studying time complexity, designating *programs* as the formal substitute to *algorithms*. The main purpose of the paper was to illustrate the convenience and usefulness of this approach. While numerous authors have already used a programming-language presentation of algorithms, the statement that the formal model of computation might be a programming language (rather than a machine) and that complexity is best evaluated this way seems to be of some novelty. The choice of a language exerts strong influence on the kind of results that may be obtained, and in this paper much consideration has been given to results that can be proved by means of an *efficient universal program*, possessed by our language. We observe that this efficiency has been obtained thanks to the fact that *the programming language is structured and so are the data*. In a way, this is just putting to use the principles which are already well known to programmers.

The approach yielded the benefits of a clearer presentation of constructions (at least to our taste) and tighter complexity results. A notable weakness of our choices regards space complexity, which is not very transparent with our language, and does not correspond closely to Turing-machine tape complexity; see van Emde Boas [36] for a discussion of this difficulty (he considers the storage modification machine, which as pointed out before is very close to the language $\mathtt{I}^{su}$).

Since there are many variations among programming languages, there is much space for exploring the use of languages as models for complexity, and we suggest that such exploration is of benefit to complexity theory and programming-language theory alike.

References

[1] Angluin D., Valiant L.: Fast probabilistic algorithms for hamiltonian circuits and matchings. J. Comput. Syst. Sci. **18**, 155–193 (1979)

[2] Balcázar J.L., Díaz J., Gabarró J.: Structural Complexity I, second edition. Berlin: Springer 1995

[3] Ben-Amram A.M.: On the power of random access machines. PhD thesis, Tel-Aviv University 1995

[4] Ben-Amram A.M.: What is a "pointer machine"? SIGACT News **26:2**, 88–95 (1995) See also `http://www.diku.dk/research-groups/topps/bibliography/1998.html#D-351`

[5] Ben-Amram A.M., Galil Z.: On pointers versus adresses. J. ACM **39:3**, 617–649 (1992)

[6] Ben-Amram A.M., Jones N.D.: A precise version of a time hierarchy theorem. Fundamenta Informaticae **38**, 1–15 (1999)

[7] Ben-Amram A.M., Petersen H.: Cons-free programs with tree input. Proceedings of ICALP 1998, Lect. Notes Comput. Sci. 1443, Springer-Verlag 1998

[8] Blum M.: A Machine-Independent Theory of the Complexity of Recursive Functions. J. ACM, **14:2**, 322–336 (1967)

[9] Cook S.A.: Linear time simulation of deterministic two-way pushdown automata. Proceedings of IFIP Congress 1971, 75–80. Amsterdam: North-Holland 1972

[10] Cook S.A., Reckhow R.A.: Time bounded random access machines. J. Comput. Sys. Sci., **7:4**, 354–375 (1973)

[11] Fürer M.: The tight deterministic time hierarchy. In Proceedings of the 14th Annual ACM Symposium on Theory of Computing, 8–16, San Francisco, California, 1982

[12] Grandjean E.: Linear time algorithms and NP-complete problems. SIAM J. Comput., **23:3**, 573–597 (1994)

[13] Gurevich Y.: Kolmogorov machines and related issues, the column on Logic in Computer Science. Bull. EATCS **35**, 71–82 (1988)

[14] Gurevich Y., Shelah S.: Nearly linear time. Logic at Botik, Lect. Notes Comput. Sci. 363, Springer-Verlag 1989

[15] Hartmanis J.: Computational Complexity of Random Access Stored Program Machines. Math. Syst. Theory **5:3**, 232–245 (1971)

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

[17] Jones N.D.: A note on linear time simulation of deterministic two-way pushdown automata. Inf. Process. Lett., **6:4**, 110–112 (1977)

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

[19] Jones N.D.: Computability and Complexity. Cambridge, Mass.: MIT Press 1997

[20] Jones N.D.: LOGSPACE and PTIME characterized by programming languages. Theor. Comput. Sci. **228**, 151–174 (1999)

[21] Kleene S.C.: Introduction to Metamathematics. Amsterdam: North-Holland 1952

[22] Knuth D.: The Art of Computer Programming, vol 1: Fundamental algorithhms. New York: Addison-Wesley 1968, 1973

[23] Levin L.A.: Universal sequential search problems (in Russian). Probl. Predachi Inf. **IX**, 115–116 (1972), English translation in Plenum Press Russian translations (1973)

[24] Luginbuhl D.R., Loui M.C.: Hierarchies and space measures for pointer machines. Information and Control **104:2**, 253–270 (1993)

[25] Pippenger N.: Pure versus Impure Lisp. ACM Transactions on Programming Languages and Systems (TOPLAS) **19:2**, 223-238 (1997)

[26] Regan K.W.: Machind models and linear time complexity, Complexity Theory Column 2 (L. A. Hemaspaandra, ed.), SIGACT News **24:3**, 5–15 (1993)

[27] Regan K.W.: Linear Speed-Up, Information Vicinity, and Finite-State Machines. In Bjorn Pehrson, Imre Simon (Eds.): Technology and Foundations — Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 1994, 609–614. North-Holland 1994

[28] Rogers H.: Theory of Recursive Functions and Effective Computability. New York: McGraw-Hill 1967

[29] Rose E.: Linear time hierarchies for a functional language machine model. In: Programming Languages and Systems – ESOP '96 (H. R. Nielson, ed.), 311–325, Lect. Notes Comput. Sci. 1058, Springer-Verlag 1996

[30] Schnorr C.P.: Optimal algorithms for self-reducible problems. Proceedings of the 3rd ICALP 1976, 322–337.

[31] Schnorr C.P.: Satisfiability is Quasilinear Complete in NQL. J. ACM **25:1**, 136–145 (1978)

[32] Schönhage A.: Storage modification machines. SIAM J. Comput. **9**, 492–508 (1980)

[33] Seiferas J.I.: Machine-independent complexity theory. *Handbook of Theoretical Computer Science*, Vol. A, 165–186, J. van Leeuwen (ed.), Amsterdam: Elsevier 1990

[34] Seiferas J.I., Fischer M.J., Meyer A.R.: Separating nondeterministic time complexity classes, J. ACM **25**, 146–167 (1978)

[35] Sudborough H., Zalcberg Z.: On families of languages defined by time-bounded random access machines. SIAM J. Comput. **5**, 217–230 (1976)

[36] van Emde Boas P.: Space measures for storage modification machines. Inf. Process. Lett. **30**, 103–110 (1989)

[37] van Emde Boas P.: Machine models and simulations. *Handbook of Theoretical Computer Science,* Vol. A, J. van Leeuwen (ed.), Amsterdam: Elsevier 1990

[38] Voda P.J.: Subrecursion as a basis for a feasible programming language. In: Proc. of the 8th workshop on computer science logic (CSL 94), Lect. Notes Comput. Sci. 933, Springer-Verlag 1995

[39] Žák S.: A Turing machine time hierarchy (note), Theor. Comput. Sci. **26**, 327–333 (1983)

```
Y := nil;
X := X.nil;      (* init stack *)
while X
   if (hd hd X) then
      (* hd X encodes a pair or is MARK *)
      if (tl hd X) then
        (* it encodes a pair; decompose *)
        X := (hd tl hd X).(tl tl hd X).MARK.(tl X)
      else
        (* it is the pair marker *)
        Y := ((nil.nil).((hd tl Y).(hd Y))).(tl tl Y);
        X := tl X
   else
      (* hd X encodes an atom *)
      if (hd X) then     (* if not nil *)
        Tmp1 := Lst;
        Tmp2 := tl hd X;
        while T2 do
          Tmp1 := tl Tmp1; Tmp2 := tl Tmp2;
        Y := Tmp1.Y
      else
        Y := nil.Y;
      X := tl X
```

Remarks: the original tree is given in X, and the resulting tree is Y. During the process, both variables are used as stacks; X for fragments of the original tree, Y for partial results. In the stack represented by X, the constant

$$\texttt{MARK} \stackrel{\text{def}}{=} ((\texttt{nil}.\texttt{nil}).\texttt{nil})$$

is used to mark the point where a pair was taken apart; note that this constant is distinguishable from encodings of both pairs and atoms.

Figure 4: Tree transformer for mau