Characterizing Computation Models with a Constant Factor Time Hierachy

Eva Rose
DIKU, University of Copenhagen*

Thursday, July 25, 1996
DIMACS Workshop
On Computational Complexity and Programming Languages

Abstract

The existence of a constant factor time hierarchy within sets decidable within a (time-constructible) time bound is a necessary property for reasoning theoretically about linear time optimizations. However, proving this property typically involves careful construction of custom interpreters or compilers, which is a tedious affair. A more generic method to state the existence or not is desirable.

In this note, we discuss some aspects of *finite branching* for a programming language (viewed as a computation model) which are necessary for the existence of a time-hierarchy assuming a realistic memory organization and implementation: a finite number of constructors, variables, and functions, and finite branching of data and control. Then, we analyse some languages for which the existence of a constant factor time-hierarchy is already known. Finally, we draw the analogy to traditional machine model theory.

1 Background

We assume the reader to be familiar with the idea of complexity from a programming perspective, the notion of a constant factor time-hierarchy, in particular related to the languages I, F and CAM (the Categorical Abstract Machine), with and without selective update facilities added [3, 9] (the latter is also available from the DIMACS workshop's electronic web-site). Basic knowledge of the Turing Machine (TM) is also assumed [4, 6].

Notice that in this note we only deal with deterministic time complexity. However, the considerations seem to generalize to non-deterministic time classes. (we refer to [3, Figure 3] for a definition of the hierarchy concept for non-deterministic time classes).

2 Analysis

In this section we will study the aspects of *branching* in a programming language and the importance of it being *finite* in order for a constant time hierarchy to exist for that language.

We assume the underlying *memory model* to be organized as an unbounded graph, where one can access the child set of a node in constant time. By "branching" we here refer to the usual branching of data values (data branching) or the branching in organization of code (code branching) in the memory. Data branching covers three aspects: the number of variables, the number of constructors, and the arity of those. Code branching on the other hand covers two aspects: the number of functions,[1] or to put it differently, the number of available "goto-labels", and the arity of the selectors (for "case" statement, for example, this is the number of entries).

First we will list and explain the language primitives which seems important for the existence of a

*Address: Universitetsparken 1, 2100 Copenhagen Ø, Denmark; email: evarose@diku.dk; WWW url: http://www.diku.dk/research-groups/topps /people/ evarose.html.

[1] This includes anonymous functions.

hierarchy:

Constructors. Data constructors are entities which construct the data values in a language. Constructors are implemented so that they can be (pairwise) tested for equality in constant time. Moreover, the retrieval of the constructor components are assumed to be performed in constant time.

The languages I, F and CAM have a fixed set of data constructors: *CONS* and *NIL*. In contrast a language like SML [7] permits definition of new data constructors within a program.

Constructor arity. The (data) arity of a constructor is the number of components that can be accessed. This can either be statically given for that language, or dynamically, depending on the program. *CONS* and *NIL* in LISP [5, 1] are examples of the static kind (with arities two and zero) wheras *Array of* or *Record* in Pascal [10] are examples of the dynamic one (with a dynamically given arity).

The maximal arity permitted for the possible constructor set in a language, defines the possible branching of a data structure. The maximal arity for I, F and CAM is two whereas that of SML is indefinite. We call this *two-way data branching* respectively ∞ *(indefinite) data branching.*

Variables. By variables we mean names that are bound to data values which can be accessed in constant time. Thus a larger number of variables in a program gives constant time access to more data.

The set of variable names which are accessible in the same amount of time, may either be given as a fixed set as in I and F, or dynamically as in LISP, SML, Haskell, or Pascal.

Functions. By functions we mean objects that can be "called" in constant time: this includes names bound to functions as well as anonymous "function values" in higher-order functional languages.

The set of such functions may either be fixed as in I and F, or not as in languages with arbitrary function (or procedure) declaration such as all usual languages.

Selectors. We hereby mean branching of flow control permitted in a control-structure, where each branch is assumed to be accessible in constant time.

The languages I, F and CAM have a fixed set of selectors, the `if`-statement.

Selector arity. The (selector) arity is given by the number of control-branches that can be accessed. This can either be statically given for that language, or dynamically, depending on the program. The `if`-statement in most languages, is an example of a statically given selector (wih arity two), whereas the `case`-statement or `goto`-statement in most languages is an example of a dynamically given one (with a dynamically given arity).

The maximal arity permitted for the possible set of selectors in a language, defines the possible branching of control-structures.

The maximal arity of I, F and CAM is two, since they only have one statically defined selector with arity two defined. The maximal arity of e.g. SML is indefinite, since `case` is a dynamically given selector with an arity given by the maximal number of allowed case entries. (For `goto`-statements, the arity is given by the set of available "jump" labels). We call this *two-way code branching* respectively ∞ *(indefinite) code branching.*

In the rest of this section, we will define the concept of a constant factor hierarchy [3] and discuss some aspects of the primitives listed above related to the existence of such a hierarchy.

Definition 1 *The class of problems decidable within time given by a total function* $f : \mathbf{N} \to \mathbf{N}$*:*

$$\begin{aligned}\textsc{Time}^L(f) \;=\; \{ & Acc^L(p) \;\mid \\ & \forall d \in L\text{-}data : time_p^L(d) \;\leq\; (f + o(f))(|d|)\ \}\end{aligned}$$

where $|d|$ *is the size of the input* d*, and* $time_p^L(d)$ *is the runtime of* L*-program* p *on input* d*.*

Definition 2 *There exists a* constant factor hierarchy within problems decidable in time f *for language* L *if and only if*

$$\exists b \geq 1\ \forall a \geq 1 : \textsc{Time}^L(a \cdot f) \subset \textsc{Time}^L(a \cdot b \cdot f)$$

We see that an (infinite) hierarchy within linear time-decidable sets is ordered by constant, multiplicative factors, that partition the set of solvable decision problems into non-empty classes.

Constant Speedup in the TM model. The *Constant Speedup theorem* [4, theorem 7.1.1] is the following.

Theorem 1 *Suppose that* $L \in \textsc{Time}(f)$*, and let* ϵ *be any positive, real number. Then* $L \in \textsc{Time}(f')$ *where for all* $n \in \mathbf{N}$ *:* $f'(n) = 2n + 18 + \lceil \epsilon \cdot f(n) \rceil$

Thus for any problem which can be solved by some TM there exists another TM which can solve the same problem linearly faster. This relies on the possibility of "extending the alphabet" of a TM at the meta-level: given a TM (that is given a concrete alphabet, state and transition descriptions), one can define another TM on the basis of the first by adding new alphabetic symbols and transition rules in such a way that the effect of processing one new symbol in one transition rule can be described as the combined effect of processing two original symbols in two original transitions. Clearly this phenomenon causes a (linear) displacement in time for essentially the same "job", a displacement which per definition cannot be registered complexity-wise at the TM meta-level. It is straigt forward to show, that the constant speedup theorem *makes any constant factor hirarchy structure collapse* (choose any $\epsilon < 1/b$).

Limiting the TM model. Hühne [2] investigates the TM model when restricting it to work on tree-structured data instead of just plain alphabetic symbols, without the usual possibility to "extend the alphabet" within the model. Instead, additional data entities are introduced (actually, encoded) through the construction of tree-structures by means of a binary data constructor. Hühne subsequently proves the Constant Speedup theorem to fail.

In a way, his TM model is slightly "lifted" towards a programming language approach. In particular we notice that the limited number of data constructors which is globally available in the Hühne TM model, correspond to limiting the alphabet in the traditional TM model.

Remark. Conceptually, this "extension of the hardware" from a programming level perspective corresponds to an extension of the available constructor set (alphabet extension) combined with an extension of the branching of the available control-structures (transition rules) in a language.

It is straight forward to prove a version of the Constant Speedup theorem, adjusted for a language which permits such extensions of its primitives.

Conclusion. The "Constant Speedup" phenomenon makes any constantly ordered hirarchy structure within a complexity class collapse.

This phenomenon can be eliminated within a language by

- limiting the available set of data constructors in a language as in the TM model of Hüne, or
- limiting the branching of control (code branching).

Data branching. Assume that a language has a fixed set of constructors available, but that one of those constructors has a dynamically given arity (as for example *Array of* in Pascal).

This means that, e.g., a nested array of some depth can be encoded as a single, concatenated array of depth one, making it possible to obtain a speedup when indexing a data element (exploiting that indexing time is independent of the array length).

Variables. Assume an unbounded number of variables is available in a language. Since this corresponds to having an implicit array structure, with an unbounded arity, available, the speedup in complexity now follows from the discussion of data branching.

Functions. Assume an unbounded number of functions is available in a language. Since this corresponds to having an implicit selector, with an unbounded arity, available, the speedup in complexity now follows from the discussion on the TM model.

3 General characterization

From the discussion above, we will first summarize the following, necessary characteristics of a pro-

gramming language in order for a constant factor hierarchy to exist (regardless of which complexity class).

1. finite set of available constructors,
2. finite branching of data.
3. finite number of variables,
4. finite number of functions, and
5. finite branching of code.

As pointed out, either limiting the set of available constructors (1), or limiting the branching in code (5), is sufficient to break the Constant Speedup phenomenon. Similarly, (5) implies (4) because only a finite number of functions can be invoked when code branching is finite. Finally, (2) implies (3) because only a finite number of variables can be accessed when data branching is finite. Hence the five characteristics may be seen as aspects of the colletive term *finite branching* used in the introduction, however, we have chosen to separate these aspects because they are usually discussed as independent characteristics of a programming language.

In the rest of this section, we will list this characterization in the case of the languages I, F, and CAM where a hierarchy is known to exist for all (time-constructible) time bounds.

Example 1 *I.*

- *two constructors (NIL,CONS)*
- *two-way data branching (CONS)*
- *one variable (*`x`*)*
- *one function (*`f`*)*
- *two-way code branching (*`if`*-statement)*

Example 2 *F.*

- *two constructors (NIL, CONS)*
- *two-way data branching (CONS)*
- *one variable (*`x`*)*
- *one function (*`f`*)*
- *two-way code branching (*`if`*-statement)*

Example 3 *CAM (categorical abstract machine).*

- *two constructors (NIL, CONS)*
- *two-way data branching (CONS)*
- *one variable (*`stack`*)*
- *one function (*`cur` *stores one "function value" on the stack)*
- *two-way code branching (*`if`*-statement)*

Remark: Adding selective update facilities has been proven not to change the existence of a hierarchy in those cases. However, the concrete hierarchy organization for linear or sublinear time-bounds seems to change, as recently pointed out by Pippenger [8]. We notice, that adding selective update facilities does not change the above characterization either.

4 Traditional computation models

Both the traditional computation models and programming languages can be regarded as computation models associated with appropriate (time) costs. For a programming language, this can be done by instrumenting its operational semantics with realistic running times.[2]

One may ask why a language-theoretical approach to complexity and computability is useful? The answer is that this is the level where a programmer is aware of optimizing effects (we also consider machine languages as programming languages). However, the correspondance to traditional computation models is, as earlier hinted, straight forward. This is illustrated in the table below, where we draw the analogy between the terminology of the Turing Machine and that of a corresponding programming languages.

Machine	*Language*
alphabetic symbol	data constructor
current state pointer	variable
transition rules	code branching
tape dimension	data branching

[2]The notion of "profiling semantics" captures the same idea.

We notice that function names are not explicitly marked in this table, since functions in a low level machine description correspond to the choice in transition rules, that is branching of code.

Example 4 TM (Turing Machine).

- **extendible alphabet** (∞ constructors).
- two dimentional tape (two-way data branching).
- current state pointer (one variable).
- **extendible set of transition rules** (∞ code branching).

5 Conclusions

We have discussed how a programming language can be characterized with respect to whether it has a constant factor time hierarchy (independent of the considered time class).

The characterization is not shown sufficient but merely a set of necessary constraints on a programming language.

References

[1] W. Clinger, J. Rees, et al. *Revised*^{4} *Report on the Algotithmic Language Scheme*, November 1991.

[2] M. Hühne. Linear speed-up does not hold on turing machines with tree storage. *Information Processing Letters*, 47:313–318, 1993.

[3] N. D. Jones. Constant time factors *do* matter. In Steven Homer, editor, *STOC '93. Symposium on Theory of Computing*, pages 602–611. ACM Press, 1993.

[4] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall Software Series. Prentice-Hall,Inc.,New Jersey, 1981.

[5] J. McCarthy. Recursive functions of symbolic expressions. *Communications of the ACM*, 3(4):184–195, April 1960.

[6] A. R. Meyer et al. *Algorithm and Complexity*, volume A of *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B.V., 1990.

[7] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[8] N. Pippenger. Pure versus impure lisp. In *POPL '96*, 1996.

[9] E. Rose. Linear time hierachies for a functional language machine mode l. In H. R. Nielson, editor, *Programming Languages and Systems – ESOP'96*, volume 1058 of *LNCS*, pages 311 – 325, Linköping, Sweden, Apr 1996. Linköping University, Springer-Verlag.

[10] N. Wirth. *The Programming Language Pascal.* ?, 1971.