Linear Time Hierarchies and Functional Languages*

Eva Rose
DIKU, University of Copenhagen†

Abstract

In STOC 93, Jones sketched the existence of a hierarchy within problems decidable in linear time by a first-order functional language based on tree-structured data (F), as well as for an extension of that language based on graph-structured data (F^{su}).

We consider the Categorical Abstract Machine (CAM), a canonical machine model for implementing higher order functional languages. We show the existence of such a hierarchy for the CAM based on tree-structured data (in particular without explicit recursion facilities), as well as in the case of graph-structured data (in particular with explicit recursion facilities). Conclusion: *first-order functional programs, and higher order functional programs all define the same class of linear-time decidable problems.*

Establishing the hierarchies for the two CAM versions, however, reveals a tension with respect to linear-time complexity hierarchies when using a computation model for higher order functional languages with selective update instructions, and one without. Conjecture: *programming with recursive data structures seems to change the linear time hierarchy.*

Keywords: linear complexity, problem hierarchy, time-decidable sets, functional languages, the categorical abstract machine, computation model, data structures, selective updating.

1 Introduction

There seems to be a gap between functional programming practice and complexity theory in general. This paper is meant to bridge some of this gap. In particular, we are concerned with whether functional programs which solve some problem have a complexity similar to programs solving the same problem in other paradigms. Clearly, studying problems with a linear time-complexity provides the most fine-grained way to study this – and, as pointed out by Jones [Jon93], the practical significance of linear time factors is wrongly underestimated, since many practical relevant decision problems can be solved in either linear or sublinear time [Reg94, PH87]

Jones [Jon94] claims the result that "*imperative programs*, *first-order functional programs*, and *higher order functional programs* all define the same class of linear time-decidable problems", and proves equivalence of the first two but not the last.

*Accepted for publication, ESOP '96.

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

The main contribution of this paper is the exposition of the last correspondence, or more precisely: we show that the same class of linear time-decidable problems is indeed defined when using a canonical machine model for higher order functional programs.

We will restrict our attention to CAM [CCM87], a canonical abstract machine for implementing higher order functional languages. The CAM is suitable for complexity considerations because it is a uniform device for measuring time complexity of different functional languages, and because it is a combinator based language, and thus operates without variables (thus side-stepping questions of variable access times).

In order to obtain the same description form of the different semantical language descriptions, we present the languages in the style of natural (operational) semantics c.f. Kahn [Kah87][1], instrumented with the assumed running times. Thus the semantics of programming languages is defined through judgements of the shape

$$\vdash \mathit{program}, \mathit{input} \stackrel{\mathit{time}}{\Longrightarrow} \mathit{value}$$

We start by an introduction to the hierarchy concept within linear time-decidable sets. The definition of an *efficient universal program* and of an *efficient interpreter* is presented in the notation used in this paper. They constitute the technical proof tools to show the existence of a hierarchy for some programming language by relating it to a known language for which the hierarchy theorem holds. We proceed by introducing a simple, first-order functional language in two versions, one which allows, and one which doesn't allow selective updating. Then we introduce CAM in two versions which basically differ on the existence of selective update facilities. Then we show our main results: there exists a hierarchy within linear time-decidable sets defined by CAM programs (abbreviated to *linear hierarchy for* CAM), one for each version of CAM. This result is established by proving the existence of efficient interpretations between the first-order functional language and CAM. However, the proof has to be done twice, one for the case of the tree-based languages, and the graph-based languages (that is possesing the `setcar!` and `setcdr!` instructions). Finally we conlude the work.

In the following, we will often write: a hierarchy for some language, when we really mean a hierarchy within the time-decidable problems defined by the programs of that language.

2 The linear time hierarchy concept

We define the concept of an (infinite) hierarchy within linear time-decidable sets, ordered by constant, multiplicative factors which partition the set of solvable decision problems into non-empty classes c.f. Jones [Jon93]. In particular, we can identify the set of problems which can be solved in linear time on a deterministic computation model by the set of deterministic[2] programs in some programming language L which encode the characteristic functions of sets and run in linear time [Pap94, M$^+$90]. Hence, a decision problem becomes a subset of the encoding programming language's data domain, and the instruction facilities in the programming language L will influence the problem-solving power, and hence the hierarchy just described. It has been shown that the linear speed-up theorem [HU79], well-known from Turing machine-theory, does not hold for tree-structured

[1] We insist on compositionality, though.

[2] By determinism, we mean no parallel facilities available in the programming language.

data [Huh93], [Jon93], since it violates the property of extending the machine alphabet without cost. A fair and realistic accounting practice is required since the time complexity measure defined on the language plays an important role for the problem solving power.

We quote Jones for the definition of a linear time-hierarchy: "There is a constant b such that for any $a \geq 1$, there is a set which is recognizable in time[3] $a \cdot b \cdot n$ but not in $a \cdot n$. Thus **LIN**, the collection of all sets recognizable in linear time by deterministic programs, contains an infinite hierarchy ordered by constant coefficients".

Another way to state this property is:

Definition 1 (linear time hierarchy) If $\exists b \forall a \geq 1 : \textsc{Time}^L(a \cdot n) \subset \textsc{Time}^L(a \cdot b \cdot n)$ then L has a linear time hierarchy where $\textsc{Time}^L(f)$ is the set of L-data which is decidable in time f by deterministic L-programs.

The constant factor b, can actually be exact determined for a hierarchy. This has e.g. been done by Hessellund and Dahl [DH94] in the case of a simple imperative language, I.

We define the notion of an efficient universal program and efficient interpretation c.f. [Jon93], adapted to the notation of this paper:

Definition 2 (efficient universal program)

- u is a *universal program* if $\forall d, p : \left[\vdash p, d \xrightarrow{t_p} v \text{ iff } \vdash u, \overline{(p, d)} \xrightarrow{t_u} v \right]$ (for some representation $\overline{\cdot}$ of programs as data).
- In particular, u is *efficient* if furthermore $\exists a \forall d, p : t_u \leq a \cdot t_p$ (where it is essential that a is independent of d and p).

Definition 3 (efficient interpretation)

- m is an *interpreter* of L written in M if $\forall d, p : \left[\vdash p, d \xrightarrow{t_p^L} v \text{ iff } \vdash m, \overline{(p, d)} \xrightarrow{t_m^M} \overline{v} \right]$ (for some representation $\overline{\cdot}$ of L-programs and values as M-data). When such an interpreter exists we write $M \succeq L$.
- In particular, m is *efficient* if furthermore $\exists e \forall d, p : t_m^M \leq e \cdot t_p^L$. This can also be formulated as: $\exists e \forall a : \textsc{Time}^L(a \cdot n) \subseteq \textsc{Time}^M(a \cdot e \cdot n)$. (where it is essential that e is independent of d and p).

3 F and F$^{\text{su}}$

We base our investigation on two Lisp-like languages defined by Jones in [Jon93] because it is known that the *constant*-or *hierarchy* theorem holds c.f. Theorem 1 below.

The languages are very restricted in that they allow only one first-order recursive function (`f`) to be defined, and only one variable name (`x`) is available, which is thus used to denote both the input to the program and the formal parameter of the function. However, mutual recursive functions as well as multiple variables can be simulated easily – and the languages are both Turing complete.

[3] n is the size of the input

The languages are strict and have running times based on standard Scheme[4] implementation technology [CR+91] (in fact they can be implemented on a unit-cost RAM in times proportional to those given here).

The languages differ in the data values that can be used: F manipulates *tree-structured datas*, that is finite cons-structures with *nil*'s for leaves. F^{su} manipulates *graph-structured datas* by allowing *selective updating* as in Scheme.

We quote [Jon93] the following Theorem on which we develop our results:

Theorem 1 (Jones, 1993) *F and F^{su} each have an efficient universal program. Further, the constant-hierarchy theorem and the efficient version of Kleene's Recursion theorem hold for F as well as for F^{su}.*

Definition 4 (Syntax, semantics and running times of F) See Figure 1.

F^{su} is defined as a store-semantic version of F, extended with `setcar!` and `setcdr!`, with the same meaning and running times as in Scheme[5]. This means that data now are *graph-structured datas*, that is either `'nil`, or possibly infinite, regular `cons`-structures with `'nil`'s as leaves for the finite branches. Since the only place where the store is updated is in the CONS-rule and in the `setcar!`, `setcdr!` rules, we omit a store description other than for these three rules.

Definition 5 (Syntax, semantics and running times of F^{su}) See Figure 2.

The special notion $\sigma@a$ is used to mean the value obtained by 'unfolding' the store tree rooted at address a – notice that this may be an infinite (regular) tree (namely when there is a cyclic path reachable from a).

4 CAM

The categorical abstract machine CAM, based on a categorical foundation [W+87], was designed with the purpose of implementing ML, an eager[6], higher order functional language. We have found 2 versions of the machine, one where recursion and branching are implicitly represented [CCM87, Table 1], and one where general recursion and branching facilities have been made explicitly available [CCM87, Table 6]. We call the first version Core-CAM; for the latter one, we use a slightly extended version, Extended-CAM, with respect to recursion operators: instead of *wind*, we allow the Lisp recursion operators *rplaca* and *rplacd*. We notice that Core-CAM manipulates tree-structured data, whereas the Extended-CAM manipulates graph-structured data.

Integers can be encoded both in F (F^{su}) and in Core-CAM (Extended-CAM) as CONS-structures, e.g. assuming a Peano-arithmetic encoding. Since these list structures have the same complexity (finite binary trees of the same depth), their encodings does not add any program dependent interpretation overhead in the proof-developments in Section 5; thus, without loss of generality we leave out integers, and operations on integers, from the CAM value domains. In Definition 6, we quote the specification of Core-CAM [CCM87, Table 1]

[4]Like traditional Lisp implementations, but with e.g. `hd 'nil` (and `tl 'nil`) defined to *nil*.

[5]The same meaning as rplaca and rplacd in traditional Lisp.

[6]We hereby understand applicative-order evaluation to weak head normal form.

Syntax

$$\begin{array}{rcl} P \in \text{Program} & ::= & E \texttt{ whererec f(x) = } E' \\ E \in \text{Expression} & ::= & \texttt{x} \mid \texttt{'nil} \mid \texttt{hd } E \mid \texttt{tl } E \mid \texttt{cons(}E'\texttt{,}E''\texttt{)} \\ & \mid & \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \mid \texttt{f(}E\texttt{)} \end{array}$$

Semantic sorts

$$d, v \in \text{Value} ::= NIL \mid CONS\ (v_1, v_2)$$

Semantic rules

$\vdash P, d \stackrel{t}{\Longrightarrow} v$: The program P, given input d, evaluates to the output v with a time cost of t.

$d, E' \vdash E \stackrel{t}{\Longrightarrow} v$: The expression E evaluates to the value v with a time cost t where the variable `x` is bound to the data structure d, and the function `f` has body E'.

$$\frac{d, E' \vdash E \stackrel{t}{\Longrightarrow} v}{\vdash E \texttt{ whererec f(x) = } E', d \stackrel{t+1}{\Longrightarrow} v} \quad \text{(F1)}$$

$$\frac{}{d, E' \vdash \texttt{x} \stackrel{1}{\Longrightarrow} d} \quad \text{(F2)}$$

$$\frac{}{d, E' \vdash \texttt{'nil} \stackrel{1}{\Longrightarrow} NIL} \qquad \frac{d, E' \vdash E_1 \stackrel{t_1}{\Longrightarrow} v_1 \quad d, E' \vdash E_2 \stackrel{t_2}{\Longrightarrow} v_2}{d, E' \vdash \texttt{cons(}E_1\texttt{,}E_2\texttt{)} \stackrel{t_1+t_2+1}{\Longrightarrow} CONS(v_1, v_2)} \quad \text{(F3, 4)}$$

$$\frac{d, E' \vdash E \stackrel{t}{\Longrightarrow} CONS(v_1, \_)}{d, E' \vdash \texttt{hd}\, E \stackrel{t+1}{\Longrightarrow} v_1} \qquad \frac{d, E' \vdash E \stackrel{t}{\Longrightarrow} NIL}{d, E' \vdash \texttt{hd}\, E \stackrel{t+1}{\Longrightarrow} NIL} \quad \text{(F5, 6)}$$

$$\frac{d, E' \vdash E \stackrel{t}{\Longrightarrow} CONS(\_, v_2)}{d, E' \vdash \texttt{tl}\, E \stackrel{t+1}{\Longrightarrow} v_2} \qquad \frac{d, E' \vdash E \stackrel{t}{\Longrightarrow} NIL}{d, E' \vdash \texttt{tl}\, E \stackrel{t+1}{\Longrightarrow} NIL} \quad \text{(F7, 8)}$$

$$\frac{d, E' \vdash v \stackrel{t_1}{\Longrightarrow} NIL \quad d, E' \vdash E_2 \stackrel{t_2}{\Longrightarrow} v_2}{d, E' \vdash \texttt{if}\, E\, \texttt{then}\, E_1\, \texttt{else}\, E_2 \stackrel{t_1+t_2+1}{\Longrightarrow} v_2} \quad \text{(F9)}$$

$$\frac{d, E' \vdash E \stackrel{t_1}{\Longrightarrow} CONS(\_, \_) \quad d, E' \vdash E_1 \stackrel{t_2}{\Longrightarrow} v_1}{d, E' \vdash \texttt{if}\, E\, \texttt{then}\, E_1\, \texttt{else}\, E_2 \stackrel{t_1+t_2+1}{\Longrightarrow} v_1} \quad \text{(F10)}$$

$$\frac{d, E' \vdash E \stackrel{t_1}{\Longrightarrow} d' \quad d', E' \vdash E' \stackrel{t_2}{\Longrightarrow} v}{d, E' \vdash \texttt{f(}E\texttt{)} \stackrel{t_1+t_2+1}{\Longrightarrow} v} \quad \text{(F11)}$$

Figure 1: F semantics

Syntax same as F but extended with

$$E \in \text{Expression} \quad ::= \quad \ldots \mid \texttt{setcar!}\ E\ E' \mid \texttt{setcdr!}\ E\ E'$$

Semantic sorts same as F but extended with

$$\begin{array}{rcl} \sigma \in \text{Store} & = & \text{Address} \to \text{Box} \\ a \in \text{Address} & = & \text{Nat} \\ \text{Box} & ::= & \mathit{NIL} \mid \mathit{CONS}\ (a_1, a_2) \end{array}$$

Semantic functions

$$\_@\_ : \text{Store} \times \text{Address} \to \text{Value} \qquad \textit{Extract value}$$

Semantic rules as F modified to use a store, in particular:

$a, E' \vdash \sigma, E \overset{t}{\Longrightarrow} \sigma', a'$ **:** The expression E evaluates in the store σ to the value in store σ' at the address a', with a time cost t, assuming `x` is bound to the value at address a in σ and `f` has body E'.

$$\frac{a, E' \vdash \sigma_0, E \overset{t}{\Longrightarrow} \sigma', a'}{E\ \texttt{whererec f(x) =}E', d \overset{t+1}{\Longrightarrow} v} \quad \sigma_0@a = d, v = \sigma'@a' \qquad (\mathrm{F^{su}}1)$$

$$\frac{a, E' \vdash \sigma, E_1 \overset{t_1}{\Longrightarrow} \sigma_1, a_1 \quad a, E' \vdash \sigma_1, E_2 \overset{t_2}{\Longrightarrow} \sigma_2, a_2}{a, E' \vdash \sigma, \texttt{setcar!}\ E_1\ E_2 \overset{t_1+t_2+1}{\Longrightarrow} \sigma_2[a_1 \mapsto \mathit{CONS}(a_2, a_1'')], a_1 \quad \textit{where } \sigma_1(a_1) = \mathit{CONS}(a_1', a_1'')} \qquad (\mathrm{F^{su}}12)$$

$$\frac{a, E' \vdash \sigma, E_1 \overset{t_1}{\Longrightarrow} \sigma_1, a_1 \quad a, E' \vdash \sigma_1, E_2 \overset{t_2}{\Longrightarrow} \sigma_2, a_2}{a, E' \vdash \sigma, \texttt{setcdr!}\ E_1\ E_2 \overset{t_1+t_2+1}{\Longrightarrow} \sigma_2[a_1 \mapsto \mathit{CONS}(a_1', a_2)], a_1 \quad \textit{where } \sigma_1(a_1) = \mathit{CONS}(a_1', a_1'')} \qquad (\mathrm{F^{su}}13)$$

Figure 2: $\mathrm{F^{su}}$ Semantics

Syntax

$$
\begin{array}{rcl}
\text{P} \in \text{Program} & ::= & \mathit{program}(\text{Cs}) \\
\text{Cs} \in \text{Commands} & ::= & \emptyset \mid \text{C}; \text{Cs} \\
\text{C} \in \text{Command} & ::= & \mathit{quote}(\alpha) \mid \mathit{car} \mid \mathit{cdr} \mid \mathit{cons} \mid \mathit{push} \mid \mathit{swap} \\
& \mid & \mathit{op}\ \textsc{plus} \mid \mathit{cur}(\text{Cs}) \mid \mathit{app}
\end{array}
$$

Semantic sorts

$$
\begin{array}{rcl}
\text{s} \in \text{Stack} & ::= & \text{s} \cdot \alpha \mid \alpha \\
\alpha, \beta \in \text{Value} & ::= & (\alpha, \beta) \mid [\text{Cs}, \alpha] \mid ()
\end{array}
$$

Semantic rules

$\vdash \mathit{program}(\text{Cs}), \alpha \stackrel{t}{\Longrightarrow} \beta$: The program $\mathit{program}(\text{Cs})$ with input α evaluates to the output β with a time cost of t

$\text{s} \vdash \text{Cs} \stackrel{t}{\Longrightarrow} \alpha$: Commands Cs evaluates to the output α with a time cost of t on input stack-value s.

$$\frac{\mathit{init\_stack} \cdot \alpha \vdash \text{Cs} \stackrel{t}{\Rightarrow} \text{s} \cdot \beta}{\vdash \mathit{program}(\text{Cs}), \alpha \stackrel{t}{\Rightarrow} \beta} \quad \text{(Cor-CAM 1)}$$

$$\frac{}{\text{s} \vdash \emptyset \stackrel{0}{\Rightarrow} S} \qquad \frac{\text{s} \vdash \text{C} \stackrel{t}{\Rightarrow} \text{s}_1 \qquad \text{s}_1 \vdash \text{Cs} \stackrel{t'}{\Rightarrow} \text{s}_2}{\text{s} \vdash \text{C}; \text{Cs} \stackrel{t+t'}{\Rightarrow} \text{s}_2} \quad \text{(Cor-CAM 2, 3)}$$

$$\frac{}{\text{s} \cdot \alpha \vdash \mathit{quote}(\alpha) \stackrel{1}{\Rightarrow} \text{s} \cdot \alpha} \quad \text{(Cor-CAM 4)}$$

$$\frac{}{\text{s} \cdot (\alpha, \beta) \vdash \mathit{car} \stackrel{1}{\Rightarrow} \text{s} \cdot \alpha} \qquad \frac{}{\text{s} \cdot (\alpha, \beta) \vdash \mathit{cdr} \stackrel{1}{\Rightarrow} \text{s} \cdot \beta} \quad \text{(Cor-CAM 5, 6)}$$

$$\frac{}{\text{s} \cdot \alpha \cdot \beta \vdash \mathit{cons} \stackrel{1}{\Rightarrow} \text{s} \cdot (\alpha, \beta)} \quad \text{(Cor-CAM 7)}$$

$$\frac{}{\text{s} \cdot \alpha \vdash \mathit{push} \stackrel{1}{\Rightarrow} \text{s} \cdot \alpha \cdot \alpha} \qquad \frac{}{\text{s} \cdot \alpha \cdot \beta \vdash \mathit{swap} \stackrel{1}{\Rightarrow} \text{s} \cdot \beta \cdot \alpha} \quad \text{(Cor-CAM 8, 9)}$$

$$\frac{}{\text{s} \cdot \rho \vdash \mathit{cur}(\text{Cs}) \stackrel{1}{\Rightarrow} \text{s} \cdot [\text{Cs}, \rho]} \qquad \frac{\text{s} \cdot (\rho, \alpha) \vdash \text{Cs} \stackrel{t}{\Rightarrow} \text{s}_1}{\text{s} \cdot ([\text{Cs}, \rho], \alpha) \vdash \mathit{app} \stackrel{t+1}{\Rightarrow} \text{s}_1} \quad \text{(Cor-CAM 10, 11)}$$

Figure 3: Core-CAM semantics

(except for the *plus*-operation as justified above), rewritten as a natural semantics [Kah87], and instrumented with assumed execution times c.f. [Han91].

Definition 6 (The Core Categorical Abstract Machine) See Figure 3.

In Definition 7, we present the Extended-CAM as a store-semantic version of Core-CAM, but extended in the way just described, and instrumented with the assumed running times. We have chosen to describe Extended-CAM by a store-semantic [Plo81] version of natural semantics in order to obtain a decompositional description of the recursion operators which also at the same time reflects the annotated running times in an intuitive way. We notice, that an *op* rule, defining the δ-rules[7] of CAM, has been added and remark that any practical use of CAM as implementation technique requires the δ-rules to be instantiated.

Definition 7 (The Extended Categorical Abstract Machine) See Figure 4.

5 A linear time hierarchy for Core-CAM

Here is our main result for Core-CAM:

Theorem 2 *There exists a linear-time hierarchy for Core-CAM.*

The proof thereof is based on the existence of efficient interpretations F $\succeq$ Core-CAM and Core-CAM $\succeq$ F.

Lemma 1 *There is an efficient interpretation F $\succeq$ Core-CAM*

Proof. An F-interpreter of Core-CAM is shown in Figure 5. To ease readability, we introduce a finite number of *atoms*: `'seq`, `'quote`, etc., to abbreviate distinct F `cons`-patterns, and some *macros*, whose expansions are explained below. At run time, the input variable is bound to $\overline{(p_c, d)}$, where p_c is some Core-CAM program and d some Core-CAM input value. The actual interpretation is performed by an interpretation loop in the LOOP macro.

The abbreviations used are the following:

- Simple definitions of the form '`LET` pattern `=` v ... `IN` E' mean that each name defined by the pattern is replaced with an appropriate decomposition of the value v inside E; furthermore `_` denotes a new name for each use. Since the decomposition is performed into a finite number of names, matching can be done within a constant time-bound.
- '`CASE` v `OF` atom$_i$ `->` E_i ... ' denotes the nested `if` statement obtained by testing the value v and selecting E_i when the value is equal to atom$_i$ (we exploit the convention of F that `'nil` = false and everything else is true). Since the number of atoms is finite, mathing can be done within a constant time-bound.
- For brevity we use E_1 . E_2 instead of the `cons(` E_1 `,` E_2 `)`.

[7] In the lambda calculus the concept means rewrite rules which are defined outside the calculus.

Syntax same as Core-CAM but extended with

$$\begin{array}{rcl} \mathrm{C} \in \mathrm{Command} & ::= & \ldots \mid \mathit{op}\ \mathrm{OP} \mid \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \\ & \mid & \mathit{rplaca} \mid \mathit{rplacd} \end{array}$$

Semantic sorts same as Core-CAM but extended with

$$\begin{array}{rcl} \alpha \in \mathrm{Value} & ::= & \ldots \mid \mathit{false} \mid \mathit{true} \\ \sigma \in \mathrm{Store} & = & \mathrm{Address} \to \mathrm{Box} \\ a, b, c \in \mathrm{Address} & = & \mathrm{Nat} \\ \mathrm{s} \in \mathrm{Stack} & ::= & \mathrm{s} \cdot a \mid a \\ \in \mathrm{Box} & ::= & (a, b) \mid [\mathrm{Cs}, a] \mid () \mid \mathit{false} \mid \mathit{true} \end{array}$$

Semantic functions

$$\_@\_ : \mathrm{Store} \times \mathrm{Address} \to \mathrm{Value} \qquad \textit{Extract value}$$

Semantic rules as Core-CAM extended with

$\mathrm{s} \vdash \sigma, \mathrm{Cs} \stackrel{t}{\Longrightarrow} \sigma', a'$: Commands Cs evaluates in the store σ to the output value at address a' in store σ' with a time cost of t on input stack-value s.

$\stackrel{\text{eval}}{\vdash}\ \mathrm{OP}, \alpha \stackrel{1}{\Rightarrow} \beta$: " eval " axiomatizes the so called δ-rules of CAM. By δ-rules, we understand rules which specify *elementary* machine operations, that is, which can be done in constant time.

$$\frac{\mathit{init\_stack} \cdot a \vdash \sigma_0, \mathrm{Cs} \stackrel{t}{\Rightarrow} \sigma, (\mathrm{s} \cdot b)}{\vdash \mathit{program}(\mathrm{Cs}), \alpha \stackrel{t}{\Rightarrow} \beta} \quad \sigma_0@a = \alpha, \beta = s'@b \qquad \text{(Ext-CAM 1)}$$

$$\frac{\stackrel{\text{eval}}{\vdash}\ \mathrm{OP}, \alpha \stackrel{1}{\Rightarrow} \beta}{\mathrm{s} \cdot \alpha \vdash \mathit{op}\ \mathrm{OP} \stackrel{1}{\Rightarrow} \mathrm{s} \cdot \beta} \qquad \text{(Ext-CAM 10)}$$

$$\frac{\mathrm{s} \vdash \mathrm{Cs}_1 \stackrel{t}{\Rightarrow} \mathrm{s}_1}{\mathrm{s} \cdot \mathit{true} \vdash \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \stackrel{t+1}{\Rightarrow} \mathrm{s}_1} \qquad \text{(Ext-CAM 13)}$$

$$\frac{\mathrm{s} \vdash \mathrm{Cs}_2 \stackrel{t}{\Rightarrow} \mathrm{s}_2}{\mathrm{s} \cdot \mathit{false} \vdash \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \stackrel{t+1}{\Rightarrow} \mathrm{s}_2} \qquad \text{(Ext-CAM 14)}$$

$$\frac{}{\mathrm{s} \cdot a \cdot b \vdash \sigma, \mathit{rplaca} \stackrel{1}{\Rightarrow} \sigma[a \mapsto \mathit{CONS}(b, a'')], \mathrm{s} \cdot a} \quad \sigma(a) = \mathit{CONS}(a', a'') \qquad \text{(Ext-CAM 15)}$$

$$\frac{}{\mathrm{s} \cdot a \cdot b \vdash \sigma, \mathit{rplacd} \stackrel{1}{\Rightarrow} \sigma[a \mapsto \mathit{CONS}(a', b)], \mathrm{s} \cdot a} \quad \sigma(a) = \mathit{CONS}(a', a'') \qquad \text{(Ext-CAM 16)}$$

Figure 4: Extended-CAM semantics

Run F-program `f(x) whererec f(x)=` LOOP on input data $\overline{p, \alpha}$, where LOOP is

```
LET  stack . (instruction . arg) = x  IN
 CASE  instruction  OF
  'empseq -> stack
  'seq ->    LET c1 . c2 = arg           IN f( f(stack . c1) . c2 )
  'quote ->  LET rest . _ = stack        IN rest . arg
  'car ->    LET rest . (a . _) = stack  IN rest . a
  'cdr ->    LET rest . (_ . b) = stack  IN rest . b
  'cons ->   LET (rest . a) . b = stack  IN rest . (a . b)
  'push ->   LET rest . a  = stack       IN rest . a . a
  'swap ->   LET (rest . b) . a = stack  IN (rest . a) . b
  'cur ->    LET rest . rho = stack      IN rest . (rho . arg)
  'app ->    LET rest . ((cs . rho) . a) = stack
                                         IN  f((rest . (rho . a)) . cs)
```

(see the text for the expansion of the `CASE`, `LET`, and *'atom* macros), and:

F-representation $\overline{\cdot}$ of Core-CAM programs:

$$\overline{(program(\mathrm{Cs}), \alpha)} = \texttt{cons(}\overline{\alpha}\texttt{,}\overline{\mathrm{Cs}}\texttt{)}$$

$$\overline{\emptyset} = \texttt{cons('empseq,'nil)} \qquad \overline{\mathrm{C};\mathrm{Cs}} = \texttt{cons('seq,cons(}\overline{\mathrm{C}}\texttt{,}\overline{\mathrm{Cs}}\texttt{))}$$

$$\overline{quote(\alpha)} = \texttt{cons('quote,}\overline{\alpha}\texttt{)}$$

$$\overline{car} = \texttt{cons('car,'nil)} \qquad \overline{cdr} = \texttt{cons('cdr,'nil)}$$

$$\overline{cons} = \texttt{cons('cons,'nil)}$$

$$\overline{push} = \texttt{cons('push,'nil)} \qquad \overline{swap} = \texttt{cons('swap,'nil)}$$

$$\overline{cur(\mathrm{Cs})} = \texttt{cons('cur,}\overline{\mathrm{Cs}}\texttt{)} \qquad \overline{app} = \texttt{cons('app,'nil)}$$

F-representation $\overline{\cdot}$ of Core-CAM values:

$$\overline{S \cdot \alpha} = \texttt{cons(}\overline{S}\texttt{,}\overline{\alpha}\texttt{)} \qquad \overline{(\alpha, \beta)} = \texttt{cons(}\overline{\alpha}\texttt{,}\overline{\beta}\texttt{)}$$

$$\overline{[\mathrm{Cs}, \alpha]} = \texttt{cons(}\overline{\mathrm{Cs}}\texttt{,}\overline{\alpha}\texttt{)} \qquad \overline{()} = \texttt{'nil}$$

Figure 5: $i_F^C(p, \alpha)$: interpreting CAM-program p on input α.

The LOOP macro represents one iteration of the interpreter hence it is easy to see by induction that *any single step of the interpreted program is realised in a bounded amount of time.* We conclude that the interpreter is efficient. □

Lemma 2 *There is an efficient interpretation Core-CAM $\succeq$ F*

Proof. A Core-CAM interpreter of F is shown in its entirety in Figure 6. The code C_{LOOP} represents one iteration of the interpreter, *consuming* one 'level' of an F-expression. Hence again it is easy to see by induction that *any single step of the interpreted program is realised in a bounded amount of time.* We conclude that the interpreter is efficient. □

Proof of Theorem 2. The above combines to $\exists e, b', f, \forall a \geq 1$:

$$\begin{aligned} \text{TIME}^C(a \cdot n) &\subseteq \text{TIME}^F(a \cdot e \cdot n) && \text{by Lemma 2} \\ &\subset \text{TIME}^F(a \cdot e \cdot b' \cdot n) && \text{by Theorem 1} \\ &\subseteq \text{TIME}^C(a \cdot e \cdot b' \cdot f \cdot n) && \text{by Lemma 1} \end{aligned}$$

Hence $\text{TIME}^C(a \cdot n) \subset \text{TIME}^C(a \cdot b \cdot n)$ with $b = e \cdot b' \cdot f$. □

6 A linear time hierarchy for the Extended-CAM

Here is our main result for Extended-CAM:

Theorem 3 *There exists a linear-time hierarchy for Extended-CAM.*

The *branch* command requires a testing facility of data structures, which cannot be encoded. Hence, for our purpose, we instantiate a δ-rule:

$$\begin{aligned} &\overset{\text{eval}}{\vdash} \text{is\_false}, false \Rightarrow true \\ &\overset{\text{eval}}{\vdash} \text{is\_false}, \alpha \Rightarrow false \qquad , \ \alpha \neq false \end{aligned}$$

Remark 1 Our *rplacd* instruction corresponds to the *wind* instruction of the original CAM [CCM87]; the symmetric *rplaca* has no counterpart in CAM. Kahn's *rec* instruction [Kah87] is similar: the simple case $rec(cur(C))$ can be simulated by the Extended-CAM code sequence:

$$push; quote(()); cons; push; cur(C); rplacd$$

Thus Extended-CAM is slightly more expresssive than both these, however not in any essential way, i.e. there does not seem to be algorithms that can be implemented more efficiently on Extended-CAM.

Lemma 3 *There is an efficient interpretation $F^{su} \succeq$ Extended-CAM*

Proof sketch. Since `setcar!`/ `setcdr!` and *rplaca*/*rplacd* implements the same operations on graph-values, it is trivial to see that the interpretation thereoff can be done efficiently. We therefore omit further details. □

Run CAM-program $program(\mathrm{C}_{\text{LOOP}})$ on input $\alpha = ((\underline{d}, \underline{E'}), \underline{E})$ where

CAM-representation $\underline{E}$ of F-expression E:

$$\begin{aligned}
\underline{\texttt{x}} &= ([\mathrm{C}_{\texttt{x}}, ()], ()) \\
\underline{\texttt{'nil}} &= ([\mathrm{C}_{\texttt{'nil}}, ()], ()) \\
\underline{\texttt{cons}(E_1, E_2)} &= ([\mathrm{C}_{\texttt{cons}}, ()], (\underline{E_1}, \underline{E_2})) \\
\underline{\texttt{hd}\ E} &= ([\mathrm{C}_{\texttt{hd}}, ()], \underline{E}) \\
\underline{\texttt{tl}\ E} &= ([\mathrm{C}_{\texttt{tl}}, ()], \underline{E}) \\
\underline{\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2} &= ([\mathrm{C}_{\texttt{if}}, ()], (\underline{E}, (\underline{E_1}, \underline{E_2}))) \\
\underline{\texttt{f(}\ E\ \texttt{)}} &= ([\mathrm{C}_{\texttt{call}}, ()], \underline{E})
\end{aligned}$$

CAM-representation $\underline{d}$ of F-value d:

$$\begin{aligned}
\underline{NIL} &= ([\mathrm{C}_{NIL}, ()], ()) \\
\underline{CONS(d_1, d_2)} &= ([\mathrm{C}_{CONS}, ()], (\underline{d_1}, \underline{d_2}))
\end{aligned}$$

CAM-code macros:

$$\begin{aligned}
\mathrm{C}_{\text{LOOP}} &= push; cdr; car; swap; push; car; swap; cdr; cdr; cons; cons; app \\
\mathrm{C}_{CONS} &= cdr; car; swap; cons; app \\
\mathrm{C}_{NIL} &= cdr; cdr; swap; cons; app \\
\mathrm{C}_{\texttt{x}} &= cdr; car; car \\
\mathrm{C}_{\texttt{nil}} &= quote(\underline{NIL}) \\
\mathrm{C}_{\texttt{cons}} &= cdr; push; push; car; swap; cdr; cdr; cons; swap; push; car; swap; cdr; car; \\
&\quad\ cons; \mathrm{C}_{\text{LOOP}}; swap; \mathrm{C}_{\text{LOOP}}; cons; push; quote([\mathrm{C}_{CONS}, ()]); swap; cons \\
\mathrm{C}_{\texttt{hd}} &= cdr; \mathrm{C}_{\text{LOOP}}; push; car; push; quote([\mathrm{C}_{\text{HTNIL}}, ()], [\mathrm{C}_{\text{HTCONS}}, ()]); cons; app; car \\
\mathrm{C}_{\texttt{tl}} &= cdr; \mathrm{C}_{\text{LOOP}}; push; car; push; quote([\mathrm{C}_{\text{HTNIL}}, ()], [\mathrm{C}_{\text{HTCONS}}, ()]); cons; app; cdr \\
&\quad\ \textit{where}\ \mathrm{C}_{\text{HTNIL}} = cdr; push; cons \\
&\quad\ \textit{and}\ \mathrm{C}_{\text{HTCONS}} = cdr; cdr \\
\mathrm{C}_{\texttt{if}} &= cdr; push; push; car; swap; cdr; cdr; cons; swap; push; car; swap; cdr; car; \\
&\quad\ cons; \mathrm{C}_{\text{LOOP}}; car; push; quote([\mathrm{C}_{\text{IFNIL}}, ()], [\mathrm{C}_{\text{IFCONS}}, ()]); cons; app \\
&\quad\ \textit{where}\ \mathrm{C}_{\text{IFNIL}} = cdr; push; car; swap; cdr; cdr; cons; \mathrm{C}_{\text{LOOP}} \\
&\quad\ \textit{and}\ \mathrm{C}_{\text{IFCONS}} = cdr; push; car; swap; cdr; car; cons; \mathrm{C}_{\text{LOOP}} \\
\mathrm{C}_{\texttt{call}} &= cdr; push; \mathrm{C}_{\text{LOOP}}; swap; car; cdr; cons; swap; car; cdr; cons; \mathrm{C}_{\text{LOOP}}
\end{aligned}$$

Figure 6: $i_C^F(p, d)$: interpreting F-program $p = E$ `whererec f(x)=` E' on input d.

Lemma 4 *There is an efficient interpretation Extended-CAM* $\succeq F^{su}$

Proof sketch. Same as the proof of 3. □

Proof sketch of theorem 3. Analogously to that of Theorem 2, with Lemma 3 replacing Lemma 1, and Lemma 4 replacing Lemma 2. □

7 Conclusions

We have shown the existence of a linear time hierarchy for Core-CAM by exposing an interpreter of Core-Cam by F and one of F by Core-CAM, based on detailed semantical descriptions of these languages, including their assumed running times, and argued why the interpretation overheads are constant. Analogously, we have shown the existence of a linear time hierarchy for Extended-CAM based on that of F^{su}.

For each of these cases we conclude that first-order functional programs and higher order functional programs define the same class of linear time-decidable problems. As a consequence, functional languages which can be implemented on CAM, have a "fair" time cost measurement.

Discussion Eventhough hierarchies exists both in the case of a tree-based functional language and in the case of a graph-based one, it seems not possible to establish a correspondance between those two data-models. In [Ros95], we study what happens when trying to both translate from the extended CAM given by Curien in [CCM87, table 6] into the F language, or to interpret this CAM version by F. Since the languages are eager, the latter forces an evaluation of an infinite graph when interpreting the recursion operator. In the first case, closures and recursive bindings intoduce program dependent slowdown proportional to the number of closure-building instructions and recursive binding instructions present in the translated program. By requiring a uniform bound on this number (as for the number of variables in the laguage), we seem to avoid this problem.

Acknowledgements Special thanks are due to my supervisor Neil Jones for introducing me to the subject and for fruitful discussions along. Also, I would like to thank Kristoffer Rose and Olivier Danvy for their thoughtful comments and encouragement.

References

[CCM87] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[CR+91] William Clinger, Jonathan Rees, et al. *Revised*^{4} *Report on the Algotithmic Language Scheme*, November 1991.

[DH94] Claus Dahl and Mogens Hessellund. Determining the constant coefficients in a time hierarchy. Student report 94-2-2, DIKU (University of Copenhagen), Department of Computer Science, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, February 1994.

[Han91] J. Hannan. Making abstract machines less abstract. In *Functional Programming Languages and Computer Architecture*, number 523 in LNCS, pages 618–635. Association for Computing Machinery, Springer-Verlag, August 1991.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley series in computer science. Addison-Wesley, 1979.

[Huh93] Martin Huhne. Linear speed-up does not hold on turing machines with tree storages. Forschungsbericht, Lehrstuhl Informatik II, Universitat Dortmund, D-W-4600 Dortmund 50, Germany, 1993.

[Jon93] Neil D. Jones. Constant time factors *do* matter. In Steven Homer, editor, *STOC '93. Symposium on Theory of Computing*, pages 602–611. ACM Press, 1993.

[Jon94] Neil D. Jones. Program speedups in theory and practice. In B. Pehrson and I. Simon, editors, *13th World Computer Congress 94*, volume 1. IFIP, Elsevier Science B.V. (North-Holland), 1994.

[Kah87] G. Kahn. Natural semantics. Rapport de Recherche 601, INRIA, Sophia-Antipolis, France, February 1987.

[M+90] A. R. Meyer et al. *Algorithm and Complexity*, volume A of *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B.V., 1990.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PH87] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. In *Lisp and Symbolic Computation.* North-Holland, 1987.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Aarhus, Denmark, 1981.

[Reg94] Ken Regan. Linear speed-up, information vicinity, and finite-state machines. In *IFIP proceedings.* North-Holland, 94.

[Ros95] Eva Rose. Linear time hierarchies and functional languages. Student report, DIKU, Department of Computer Science, Universitetsparken 1, 2100 Copenhagen Ø, Denmark, 1995.

[W+87] P. Weis et al. *The CAML Reference Manual.* INRIA-ENS, version 2.5 edition, December 1987.