Linear Time Hierarchies for a Functional Language Machine Model

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 (without selective updating facilities), as well as in the case of graph-structured data (with selective updating). In conclusion we establish two local robustness results where *first-order functional programs and higher order functional programs define the same class of linear-time decidable problems.*

Keywords: linear time, complexity hierarchy, CAM, operational semantics, functional languages, selective update, structured data.

1 Introduction

There seems to be a gap between functional programming practice and complexity theory. This paper is meant to bridge some of this gap. In particular, we are concerned with the question of 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 perspective on this – and, as pointed out by Jones [Jon93], the practical significance of constant time factors is wrongly underestimated, since many practical relevant decision problems can be solved in linear 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

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

not the last. The 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 restrict our attention to CAM [CCM87], a canonical abstract (environment) machine [Cur90] for implementing higher order functional languages. The CAM is suitable for complexity considerations for two reasons. First, because it is a uniform device for measuring running times of different functional languages. Second, because it is a combinator based language, e.g., operates without variables, thus side-stepping questions of variable access times.

Another issue is the widely believed conjecture that the presence of selective updating (hence cyclic graphs) makes the computational model stronger in an asymptotic sense [BA95]. In support of this we have found it necessary to show the hierarchy property seperately for languages with and without such updating.

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

$$\vdash \textit{program}, \textit{input} \overset{\textit{time}}{\Longrightarrow} \textit{value}$$

In Section 2, we start out with an introduction to the hierarchy concept within linear time-decidable sets. In particular the definition of an *efficient interpreter* is presented in the notation used in this paper. In Section 3, we proceed by introducing a simple, first-order functional language in two versions: one which allows selective updating, and one which does not (in concordance with the above considerations). In Section 4, we introduce the CAM in two similar versions. In Section 5 and 6 we show our results: there exists a hierarchy within linear time-decidable sets defined by CAM programs, one for each version of CAM. Finally, we conclude the work.

2 The linear time hierarchy concept

Taking a programming language approach to complexity implies identifying an *algorithm* by a *program*. This identifies the set of problems which can be solved on a deterministic computation model with the set of deterministic[2] programs of some programming language, L, which encode the characteristic functions [Pap94, M^+90]. Hence, a decision problem becomes a subset of the encoding programming language's data domain, L-data. The following definitions 1–4 and 6 are adopted from [BAJ95].

Definition 1 *Any L-program, p, represents a decision problem:*

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

[1] We insist on compositionality, though.

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

Definition 2 *The class of problems decidable within time given by a total function $f : \mathrm{N} \to \mathrm{N}$:*

$$\mathrm{TIME}^L(f) = \{Acc^L(p) \mid \forall d \in L\text{-}data : \ time_p^L(d) \ \leq \ (f + o(f))(|d|) \ \}$$

where $|d|$ is the size of the input d, and $time_p^L(d)$ is the runtime, determined by L's instrumented operational semantics.

We recall the definition of $o(f)$:

$$o(f(n)) \ = \ \epsilon(n) \times f(n) \text{ where } \epsilon(n) \to 0 \text{ for } n \to \infty$$

Definition 3 *The class of linear time decidable problems given by a total function $\ell_a : \mathrm{N} \to \mathrm{N}$ defined by $\ell_a(n) = an$ for any $n \in \mathrm{N}$:*

$$LIN^L(a) = \mathrm{TIME}^L(\ell_a)$$

Following [Jon93], we can now, in the formalism just quoted, define the concept of an (infinite) hierarchy within linear time-decidable sets, ordered by constant, multiplicative factors, that partition the set of solvable decision problems into non-empty classes:

Definition 4 *There exists a* hierarchy within problems decidable in linear time *by language L if and only if*

$$\exists b \geq 1 \, \forall a \geq 1 : LIN^L(a) \subsetneq LIN^L(a \cdot b)$$

The constant factor b, can actually be exactly determined for a concrete hierarchy, e.g., Hessellund and Dahl determined it to be at least 249 in the case of a simple imperative language I [DH94].

We need a notion of *representation* to be able to relate the program and data terms of different languages. However, we have to be careful that the representation does not allow nontrivial encodings, e.g. (p, d) as p paired with the result of running p on d.

Definition 5 *A map from one set of terms T_1 to another T_2, $\overline{\cdot} : T_1 \to T_2$, is a* representation *if it is defined compositionally over the syntactic structure of T_1 such that the number of composition-steps is bounded by the depth of the term.*

We now define the notion of an efficient interpretation (c.f. [Jon93]) adapted to the notation of this paper and our more general notion of representation:

Definition 6 (efficient interpretation)

- m is an *interpreter* of L written in M if $\forall d, p$:

  $$\vdash p, d \xrightarrow{\mathrm{time}_p^L} v \text{ iff } \vdash m, (\overline{p}, \overline{d}) \xrightarrow{\mathrm{time}_m^M} \overline{v}$$

  for some representation, $\overline{\cdot}$, of L-programs and L-data as M-data, assuming that a pair/cons-operation in language M takes constant time. When such an interpreter exists we write $M \succeq L$.

- In particular, m is *efficient* iff $\exists e \geq 1\ \forall d, p : \text{time}_m^M \leq e \cdot \text{time}_p^L$. Provided that L-data and M-data are defined over the same domain,[3] and time_p^L bounded by some linear ℓ_a, with $a \geq 1$, this can be formulated as:

$$\exists e, a \geq 1 : LIN^L(a) \subseteq LIN^M(e \cdot a)$$

(where it is essential that e is independent of d and p).

3 F and F^{su}

We base our investigation on two Lisp-like languages defined by Jones [Jon93] because it is known that the *constant-* or *hierarchy* theorem holds, see Theorem 1 below. We present the language definitions as natural semantics in Figure 1 and Figure 2, instrumented with realistic running times. 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`), 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. The languages are strict and have running times based on standard Scheme[4] implementation technology [CR$^+$91] (in fact they can be implemented on a unit-cost RAM in times proportional to those given here). Basically, they differ in the data values on which they operate: F manipulates *tree-structured data*, i.e. finite, directed trees, with "NIL" for leaves, and whose internal nodes, the "$CONS$-cells", each have out-degree two. F^{su}, however, manipulates *graph-structured data* by allowing *selective updating* as in Scheme. Graph-structured data are defined as finite and directed graphs in the sense of Barendregt et.al. [BvEG$^+$87] with leaves labelled "NIL", and where the internal nodes, labelled "$CONS$", have out-degree two; further, each node is identified by a unique number. In the following, graph-structured data are called 'boxes', and each node-identifier, a 'location'. We notice that the definition allows cyclic paths in the graph. We now quote from [Jon93] the 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 the Kleene recursion theorem hold for F as well as for F^{su}.*

Definition 7 (Syntax, semantics and running times of F) in Figure 1.

Note that we have exploited the fact that in F there are always exactly the two bindings of the symbols `x` and `f` in the 'environment', which we have therefore marked implicitly. Instead of $[\texttt{x} \mapsto d\ ;\ \texttt{f} \mapsto E']$ we simply write d, E'.

[3]Actually, a structure-preserving isomorphism between them is sufficient.
[4]Like traditional Lisp implementations, but with e.g. `hd 'nil` (and `tl 'nil`) defined to *nil*.

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

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

Semantic rules

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

$d, E' \vdash E \overset{t}{\Longrightarrow} v$: 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'.

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

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

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

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

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

$$\frac{d, E' \vdash E \overset{t_1}{\Longrightarrow} \mathit{CONS}\,(\_, \_) \quad d, E' \vdash E_1 \overset{t_2}{\Longrightarrow} v_1}{d, E' \vdash \texttt{if}\,E\,\texttt{then}\,E_1\,\texttt{else}\,E_2 \overset{t_1+t_2+1}{\Longrightarrow} v_1} \quad \text{(F9)}$$

$$\frac{d, E' \vdash E \overset{t_1}{\Longrightarrow} \mathit{NIL} \quad d, E' \vdash E_2 \overset{t_2}{\Longrightarrow} v_2}{d, E' \vdash \texttt{if}\,E\,\texttt{then}\,E_1\,\texttt{else}\,E_2 \overset{t_1+t_2+1}{\Longrightarrow} v_2} \quad \text{(F10)}$$

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

Figure 1: F semantics and running times.

F^{su} is defined as a store-based version of F following Plotkin [Plo81], extended with **setcar!** and **setcdr!**, with the same meaning and running times as in Scheme[5]. This means that the variable binding description becomes a two-level description, introducing graph-structured data (*boxes*) as the intermediate step. Hence, the variable binding $\mathtt{x} \mapsto v$ becomes $\mathtt{x} \mapsto l \overset{\sigma}{\mapsto} v$ in F^{su}, where l is a *location*[6] and σ is a *store*, mapping locations to boxes (where a location identifies the root of its box in that store). We introduce a special notation, a partial function, $\sigma@l$, to denote the tree-structured value obtained by unravelling the box $\sigma(l)$ from its root l in the store σ. We notice that $\_@\_$ is only defined when no cyclic paths are reachable from l.

Definition 8 (Syntax, semantics and running times of F^{su}) in Figure 2.

We notice, that the bindings of the symbols x and f in the 'extended environment' has also been made implicit. Instead of $[\mathtt{x} \mapsto l \ ; \mathtt{f} \mapsto E']$ we simply write l, E'. Also, we remark that the constant location, $[l_{\text{nil}} \mapsto NIL]$, is invariantly part of any store since it is part of the initial one, σ_0.

The only place where the store is updated is in the CONS-rule (where a new memory location, l_{fresh}, can be allocated in constant time) and in the **setcar!**, **setcdr!** rules, where cyclic structures might be introduced. Hence only these rules have been explicitly stated. We notice that F and F^{su} correspond through the relations: $\sigma@l = d$, $\sigma'@l' = v$ exactly when a program is terminating.

4 CAM

Our target machine is the environment-based, categorical abstract machine CAM, developed on a categorical foundation by Cousineau, Curien, Mauny [CCM87]. Its instructions form a fixed set of (categorical) combinators, constructed to be faithful to β-reduction in the λ-calculus, and acting on a graph-environment (stack). It is the binding-height which defines a variable binding – since no variables are explicit in the model. As described in [Jon93], it is essential for program independent interpretation, that the number of variable names is bounded. This is why we approach a model like CAM (and the reason for which we cannot approach higher-order functional languages in general). The CAM implements a call-by-value evaluation strategy, and is suitable for implementing ML, an eager[7], higher-order functional language [CCM87],[W$^+$87]. Originally there are two versions: one where recursion and branching are implicitly represented [CCM87, Table 1], hence operating on tree-structured values, and one where general recursion and branching facilities have been made explicit [CCM87, Table 6], that is working on graph-structured values. We use this classification for our CAM versions: *Core-CAM* in the first case, *Ext-CAM* in the latter. However, we present the languages as natural semantics following Kahn [Kah87]. Actually, Ext-CAM

[5]The same meaning as rplaca and rplacd in traditional Lisp.
[6]Following Plotkin, a location is independent of any concrete machine technology.
[7]We hereby understand applicative-order evaluation to weak head normal form.

Syntax same as F but extended with

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

Semantic sorts same as F but extended with

$$\begin{array}{rcl} \sigma \in \text{Store} & = & \text{Location} \rightharpoonup \text{Box} \\ l \in \text{Location} & = & \text{Nat} \\ \text{Box} & ::= & \mathit{NIL} \mid \mathit{CONS}\,(l_1, l_2) \end{array}$$

Semantic functions

$\_@\_ : \text{Store} \times \text{Location} \rightharpoonup \text{Value}$ *Extract value (partial function)*

Semantic rules as F modified to use a store:

$l, E' \vdash \sigma, E \overset{t}{\Longrightarrow} \sigma', l'$: The expression E evaluates in store σ to location l' and store σ', with a time cost t, assuming $\mathbf{x}$ is bound to location l and $\mathbf{f}$ is bound to E'.

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

$$\textit{where } \sigma_0 @\, l_{\text{nil}} = \mathit{NIL},\ \sigma_0 @\, l = d,\ \sigma' @\, l' = v$$

$$\frac{l, E' \vdash \sigma, E_1 \overset{t_1}{\Longrightarrow} \sigma_1, l_1 \quad l, E' \vdash \sigma_1, E_2 \overset{t_2}{\Longrightarrow} \sigma_2, l_2}{l, E' \vdash \sigma, \texttt{cons(}\, E_1\, \texttt{,}\, E_2\, \texttt{)} \overset{t_1+t_2+1}{\Longrightarrow} \sigma_2[\, l_{\text{fresh}} \mapsto \mathit{CONS}\,(l_1, l_2)\,],\ l_{\text{fresh}}} \qquad (\mathrm{F^{su}4})$$

$$\textit{where } l_{\text{fresh}} \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{l, E' \vdash \sigma, E_1 \overset{t_1}{\Longrightarrow} \sigma_1, l_1 \quad l, E' \vdash \sigma_1, E_2 \overset{t_2}{\Longrightarrow} \sigma_2, l_2}{l, E' \vdash \sigma, \texttt{setcar!}\ E_1\ E_2 \overset{t_1+t_2+1}{\Longrightarrow} \sigma_2[\, l_1 \mapsto \mathit{CONS}\,(l_2, l_1'')\,],\ l_1} \qquad (\mathrm{F^{su}12})$$

$$\textit{where } \sigma_1(l_1) = \mathit{CONS}\,(l_1', l_1'')$$

$$\frac{l, E' \vdash \sigma, E_1 \overset{t_1}{\Longrightarrow} \sigma_1, l_1 \quad l, E' \vdash \sigma_1, E_2 \overset{t_2}{\Longrightarrow} \sigma_2, l_2}{l, E' \vdash \sigma, \texttt{setcdr!}\ E_1\ E_2 \overset{t_1+t_2+1}{\Longrightarrow} \sigma_2[\, l_1 \mapsto \mathit{CONS}\,(l_1', l_2)\,],\ l_1} \qquad (\mathrm{F^{su}13})$$

$$\textit{where } \sigma_1(l_1) = \mathit{CONS}\,(l_1', l_1'')$$

Figure 2: $\mathrm{F^{su}}$ semantics and running times.

has been slightly extended: the original *wind*-instruction is replaced by the identically defined *rplacd*,[8] and we add its symmetrical instruction, *rplaca*, which has no counterpart in CAM originally; this is of no complexity-consequence since the one can simulate the other efficiently (see Rose [Ros96]).

[8]Kahn's recursion operator *rec* [Kah87], is essentially defined in terms of *rplacd*.

Syntax

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

Semantic sorts

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

Semantic rules

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

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

$$\frac{() \cdot \alpha \vdash \mathrm{Cs} \overset{t}{\Rightarrow} \mathrm{s} \cdot \beta}{\vdash \mathit{program}(\mathrm{Cs}), \alpha \overset{t}{\Rightarrow} \beta} \quad \text{(C-CAM 1)}$$

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

$$\frac{}{\mathrm{s} \cdot \beta \vdash \mathit{quote}(\alpha) \overset{1}{\Rightarrow} \mathrm{s} \cdot \alpha} \quad \text{(C-CAM 4)}$$

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

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

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

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

Figure 3: Core-CAM semantics and running times.

To ease the proof developments, we omit integers and integer operations since they can be encoded in F (F^{su}) and in Core-CAM (Ext-CAM) in the same way (with respect to complexity) e.g. as Church numerals or using Peano arithmetic. We present Core-CAM in Figure 3, and Ext-CAM in Figure 4, instrumented with the assumed execution times. These are based on an analysis of CAM by Hannan [Han91] (for details refer to Rose [Ros96]).

Definition 9 (Syntax, semantics and running times of Core-CAM) in Figure 3.

We notice that the constant locations, $[l_{()} \mapsto ()]$, is invariantly part of any store since it is part of the initial, σ_0.

In Definition 10, we present the Extend-CAM as a store-semantic version of Core-CAM following Plotkin [Plo81]. We have adapted the notation from section 3.

Definition 10 (Syntax, semantics and running times of Ext-CAM) in Figure 4.

Note that the constant locations $[l_{()} \mapsto (), l_{\text{false}} \mapsto \mathit{false}, l_{\text{true}} \mapsto \mathit{true}]$ are invariantly part of any store since they are part of the initial store σ_0. The *quote* rule (E-CAM 4) deserves special mention. Its purpose is to add a value to the store. In Core-CAM without selective updating, this can be done in time 1 because constant values remain constant. However, in Ext-CAM a $\mathit{quote}(\alpha)$ command takes time $|\alpha|$ since the model must allocate a fresh copy each time (this is represented by the requirement that $(\sigma_1 \backslash \sigma) @ l_1 = \alpha$) to allow selective updating of this copy without destroying any data (this is represented by $\sigma \subsetneq \sigma_1$ which incidentally implies that we cannot do "garbage collection"). In analogy with F^{su}, we only list those rules which have an effect on the store.

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 $(\overline{p_c}, \overline{d})$, 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; since we only decompose subterms of the original value this cannot result in code duplication. 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.

Syntax same as Core-CAM but extended with

$$\mathrm{C} \in \text{Command} \ ::= \ \ldots \mid \mathit{op}\ \text{is\_false} \mid \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \mid \mathit{rplaca} \mid \mathit{rplacd}$$

Semantic sorts same as Core-CAM but extended with

$$\begin{array}{rcl} \alpha \in \text{Value} & ::= & \ldots \mid \mathit{false} \mid \mathit{true} \\ \sigma \in \text{Store} & = & \text{Location} \rightharpoonup \text{Box} \\ l \in \text{Location} & = & \text{Nat} \\ \mathrm{s} \in \text{Stack} & ::= & \mathrm{s} \cdot l \mid l \\ \text{Box} & ::= & (l_1, l_2) \mid [\mathrm{Cs}, l] \mid () \mid \mathit{false} \mid \mathit{true} \end{array}$$

Semantic functions

$\_@\_ : \text{Store} \times \text{Location} \rightharpoonup \text{Value}$ *Extract value (partial function)*

Semantic rules as Core-CAM modified to use a store:

$\mathrm{s} \vdash \sigma, \mathrm{Cs} \overset{t}{\Longrightarrow} \sigma', \mathrm{s}'$: Commands Cs evaluates on stack s in store σ to the store σ' and output stack s′ with a time cost of t.

$$\frac{() \cdot l \vdash \sigma_0, \mathrm{Cs} \overset{t}{\Rightarrow} \sigma', \mathrm{s} \cdot l'}{\vdash \mathit{program}(\mathrm{Cs}), \alpha \overset{t}{\Rightarrow} \beta} \quad \sigma_0 @ l = \alpha,\ \sigma' @ l' = \beta \qquad \text{(E-CAM 1)}$$

$$\textit{where } \sigma_0 @ l_{\mathit{false}} = \mathit{false},\ \sigma_0 @ l_{\mathit{true}} = \mathit{true},\ \sigma_0 @ l_{()} = ()$$

$$\frac{}{\mathrm{s} \cdot l \vdash \sigma, \mathit{quote}(\alpha) \overset{|\alpha|}{\Rightarrow} \sigma_1, \mathrm{s} \cdot l_1} \quad \sigma \subseteq \sigma_1,\ (\sigma_1 \backslash \sigma) @ l_1 = \alpha \qquad \text{(E-CAM 4)}$$

$$\frac{}{\mathrm{s} \cdot l_1 \cdot l_2 \vdash \sigma, \mathit{cons} \overset{1}{\Rightarrow} \sigma[l_{\text{fresh}} \mapsto (l_1, l_2)],\ \mathrm{s} \cdot l_{\text{fresh}}} \quad l_{\text{fresh}} \notin \mathrm{Dom}(\sigma) \qquad \text{(E-CAM 7)}$$

$$\frac{}{\mathrm{s} \cdot l\ \vdash \sigma, \mathit{cur}(\mathrm{Cs}) \overset{1}{\Rightarrow} \sigma[l_{\text{fresh}} \mapsto [\mathrm{Cs}, l]], \mathrm{s} \cdot l_{\text{fresh}}} \quad l_{\text{fresh}} \notin \mathrm{Dom}(\sigma) \qquad \text{(E-CAM 10)}$$

$$\frac{\mathrm{s} \vdash \sigma, \mathrm{Cs}_1 \overset{t}{\Rightarrow} \sigma_1, \mathrm{s}_1}{\mathrm{s} \cdot \mathit{true} \vdash \sigma, \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \overset{t+1}{\Rightarrow} \sigma_1, \mathrm{s}_1} \qquad \text{(E-CAM 13)}$$

$$\frac{\mathrm{s} \vdash \sigma, \mathrm{Cs}_2 \overset{t}{\Rightarrow} \sigma_1, \mathrm{s}_2}{\mathrm{s} \cdot \mathit{false} \vdash \sigma, \mathit{branch}(\mathrm{Cs}_1, \mathrm{Cs}_2) \overset{t+1}{\Rightarrow} \sigma_1, \mathrm{s}_2} \qquad \text{(E-CAM 14)}$$

$$\frac{}{\mathrm{s} \cdot l \cdot l_2 \vdash \sigma, \mathit{rplaca} \overset{1}{\Rightarrow} \sigma[l \mapsto \mathit{CONS}(l_2, l'')], \mathrm{s} \cdot l} \quad \sigma(l) = \mathit{CONS}(l', l'') \qquad \text{(E-CAM 15)}$$

$$\frac{}{\mathrm{s} \cdot l \cdot l_2 \vdash \sigma, \mathit{rplacd} \overset{1}{\Rightarrow} \sigma[l \mapsto \mathit{CONS}(l', l_2)], \mathrm{s} \cdot l} \quad \sigma(l) = \mathit{CONS}(l', l'') \qquad \text{(E-CAM 16)}$$

$$\frac{}{\mathrm{s} \cdot l \vdash \sigma, \mathit{op}\ \text{is\_false} \overset{1}{\Rightarrow} \sigma[l \mapsto \mathit{true}],\ \mathrm{s} \cdot l} \quad \sigma @ l = \mathit{false} \qquad \text{(E-CAM 17)}$$

$$\frac{}{\mathrm{s} \cdot l \vdash \sigma, \mathit{op}\ \text{is\_false} \overset{1}{\Rightarrow} \sigma[l \mapsto \mathit{false}],\ \mathrm{s} \cdot l} \quad \sigma @ l \neq \mathit{false} \qquad \text{(E-CAM 18)}$$

Figure 4: Ext-CAM semantics and running times.

Run F-program

$$\texttt{f(tl x,hd x) whererec f(x)} = \textsc{Loop}$$

on input data $\texttt{cons(}\overline{\texttt{P}}\texttt{,}\overline{\alpha}\texttt{)}$, 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-IN**, ., and *'atom* macros).

Invariant: $(\overline{\alpha},\overline{C}), \textsc{Loop} \vdash_{\overline{F}} \textsc{Loop} \Longrightarrow \overline{\beta}$ iff $\alpha \vdash_{\overline{CAM}} C \Longrightarrow \beta$

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

$$\overline{program(\mathrm{Cs})} = \overline{\mathrm{Cs}}$$

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

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

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

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

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

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

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

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: $i^{C}_{F}(p,\alpha)$: interpreting CAM-program p on input α.

- '`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 `cons(`E_1 `,` E_2`)`.

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 $\mathrm{C}_{\mathrm{LOOP}}$ represents one iteration of the interpreter, *consuming* one 'level' of an F-expression. 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 $\exists e, b', c, \forall a \geq 1$:

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

Hence $LIN^C(a \cdot n) \subsetneq LIN^C(a \cdot b \cdot n)$ with $b = e \cdot b' \cdot c$. □

6 A linear time hierarchy for the Ext-CAM

Here is our main result for Ext-CAM:

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

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

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

Lemma 4 *There is an efficient interpretation Ext-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. □

Run CAM-program $\mathit{program}(\mathrm{C}_{\text{INIT}}; \mathrm{C}_{\text{LOOP}})$ on input $\alpha = (\underline{P}, \underline{d})$ where

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

Invariant: $S \cdot ((\underline{d}, \underline{E'}), \underline{E}) \vdash_{\overline{CAM}} \mathrm{C}_{\text{LOOP}} \Longrightarrow S \cdot \underline{v}$ iff $d, E' \vdash_{\overline{F}} E \Longrightarrow v$

CAM-representation $\underline{\cdot}$ of F-program/expression P/E:

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

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

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

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

7 Conclusions

We have shown the existence of a linear time hierarchy for Core-CAM through exposition of an efficient interpreter of Core-Cam by F, and an efficient interpreter of F by Core-CAM. Similarly, we have argued for the existence of a linear time hierarchy for Ext-CAM by efficient interpretation to and from F^{su}. Thus we have established that LIN is robust with respect to transition between first and higher order functional programming models (this is interesting because LIN is not generally robust [GS85]).

Acknowledgements. Thanks are due to my supervisor, Neil Jones, for introducing me to the subject and for fruitful discussions along. Special thanks go to Kristoffer Rose, Olivier Danvy, Amir Ben-Amram, Morten Sørensen, and in particular Peter Sestoft, for their valuable comments.

References

[BA95] A. M. Ben-Amram. Pointer machines and pointer algorithms: an annotated bibliography. Diku-rapport 95/21, DIKU (Department of Computer Science), University of Copenhagen, September 1995.

[BAJ95] A. M. Ben-Amram and N. D. Jones. Complexity-theoretic advantages of structured programs and structured data. Personal communication, October 1995.

[BvEG+87] H. P. Barendregt, M. C. D. J. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87—Parallel Architectures and Languages Europe vol. II*, number 256 in LNCS, pages 141–158, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

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

[CR+91] W. Clinger, J. Rees, et al. *Revised^4 Report on the Algotithmic Language Scheme*, November 1991.

[Cur90] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1990.

[DH94] C. Dahl and M. 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.

[GS85] Y. Gurevich and S. Shelah. Nearly linear time. In *Logic at Botik*, volume 363 of *LNCS*, pages 108–118. Springer-Verlag, 1985.

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

[Jon93] 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.

[Jon94] N. 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] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PH87] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. In *Lisp and Symbolic Computation*, volume 4, pages 207–232. North-Holland, August 1987.

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

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

[Ros96] E. Rose. Linear time hierarchies for a functional language machine model. Student report, DIKU, Department of Computer Science, Universitetsparken 1, 2100 Copenhagen Ø, Denmark, 1996.

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